A DESIGN FOR A FUNCTION-DESCRIPTIVE PROGRAMMING LANGUAGE

Jerry Gregory Paccassi

NPS52-78-003

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A DESIGN FOR A
FUNCTION-DESCRIPTIVE PROGRAMMING LANGUAGE

by

Jerry Gregory Paccassi II

and

Carl Eric Wick

June 1978

Thesis Advisor: U.R. Kodres

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral Tyler Dedman
Superintendent

Jack R. Borsting
Provost

Released as a
Technical Report by:

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: NPS52-78-003
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): A Design for a Function-Descriptive Programming Language
5. TYPE OF REPORT & PERIOD COVERED: Master's Thesis; June 1978
6. PERFORMING ORG. REPORT NUMBER:

7. AUTHOR(s): Jerry Gregory Paccassi II
Carl Eric Wick

8. CONTRACT OR GRANT NUMBER(s):

9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School
Monterey, California 93940

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:

11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School
Monterey, California 93940

12. REPORT DATE: June 1978
13. NUMBER OF PAGES: 227

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):

15. SECURITY CLASS. (of this report): Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Programming Language
Function-Descriptive Programming Language

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A design for a function-descriptive programming language is described. The language is based upon a software design model which uses the process of abstraction and successive refinement in problem solving. The resulting programming language provides mechanisms and structures conducive to language extension, ease of program development and enhancement of software reliability. A system's library expands the

(20. ABSTRACT Continued)

capability of the base language to satisfy the needs of a user or user group. The user, therefore, does not need to carry the burden of those features of the language which he does not use. Because of its potentially small size, the translator or compiler of the base language may be feasibly implemented on microcomputer developmental systems.

Approved for public release; distribution unlimited.

A Design for a Function-Descriptive Programming Language

by

Jerry Gregory Paccassi II
Major, United States Marine Corps
B.M.E., University of Santa Clara, 1967

and

Carl Eric Wick
Lieutenant, United States Navy
B.S., United States Naval Academy, 1970

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

June 1978

ABSTRACT

A design for a function-descriptive programming language is described. The language is based upon a software design model which uses the process of abstraction and successive refinement in problem solving. The resulting programming language provides mechanisms and structures conducive to language extension, ease of program development and enhancement of software reliability. A system's library expands the capability of the base language to satisfy the needs of a user or user group. The user, therefore, does not need to carry the burden of those features of the language which he does not use. Because of its potentially small size, the translator or compiler of the base language may be feasibly implemented on microcomputer developmental systems.

TABLE OF CONTENTS

ACKNOWLEDGEMENT

The authors wish to express their sincere gratitude to Dr. Uno Kodres, who, because of his great energy and unselfish desire to provide timely advice and counsel in student research, is largely responsible for the success of this work. Thanks are also due to Mrs. Rosemary Lande, typist, whose talents were instrumental in the production of the final version of this text. Lastly, but certainly not least, our sincere appreciation is due to our wives for their untiring patience, support and eager assistance in typing, proofreading and providing suggestions.

I. INTRODUCTION

High level programming languages have proven to be an important component of modern software development. Programming languages, in their various forms, are powerful tools to be used in projecting the design of a computer application into a useful product. All programming languages have the fundamental property that concepts and procedures are recorded in a form which is both humanly readable and is translatable by a computer into a set of machine executable instructions. This set of instructions, when executed by a computer, ideally reflects the intent of its originator.

Primitive forms of programming languages began appearing soon after the introduction of the electronic computer. From these primitive origins has grown and evolved a family of programming languages with much variety. As has been generally true in the process of biological evolution, it could be expected that succeeding generations of these languages would be better suited to their tasks. There is, however, some question as to whether this has happened. In their paper which reviews language designs, Richard and Ledgard said:

> "most existing programming languages do not suit the production of large software systems. To implement real problems, no current programming language offers clear solutions... .
>
> In our opinion, current programming languages are seriously afflicted and a radical cure is in order." [1, p. 73]

The new generation of high-level programming languages, as observed in the preliminary report of the Department of Defense Common High Order Language Evaluation Committee proposals for a new computer language DOD-1 [19], does not appear to be rapidly evolving from present forms into integrated members of software engineering technology. Until a substantial improvement in programming language takes place which is suitable for the entire software development cycle, complexity, errors and problems of system maintenance are likely to remain as costly factors in software. The rapidly growing expenses of software force us to consider, as the remarks of Richard and Ledgard suggest, a "radical cure." Such a language should be an image of the overall design system it is used in. It should also allow the use of uniform design procedures from macroscopic design conceptions down to the least element of program detail. Further, such a language system should be built to provide a common base for the program users, the designers, and programmers who solve the user requirements. A language should allow the user to participate more fully, and with less confusion, in the construction of software systems. Benefits of this approach would be fewer unfulfilled user requirements and fewer overall difficulties arising from multilevel inter-personal communications.

This thesis presents the design of a programming language which allows functional problem description in a top-down manner. The language is designed to provide for

generality of application and is based on the concepts and principles of software engineering and functional abstraction. The thesis is organized in three major sections. Section II provides background information and factors which motivated this research. Section III discusses the philosophy which formed the basis of design of this functionally oriented language. Section IV is an overview which highlights the kernel of the produced function-descriptive language design and a complete description of the syntax of the product language. Syntax diagrams were used as a pictorial vehicle to present clearly to the reader both language syntax and structure.

II. BACKGROUND

A. EVOLUTION OF SOFTWARE GOALS

Historically, hardware characteristics of computers have provided a large contribution to the goals of early software design. These characteristics, through the establishment of goals, have also influenced the construction of succeeding generations of programming languages and have been a directing factor in the path of their evolution. Aron [2] and Sammet [3] described the nature of these environmental forces during the earlier days of electronic computation. The first machines were ponderous, slow, relatively unreliable and had little computing capacity. Time and space constraints caused by hardware configuration, coupled with the tremendous chore of programming computers in absolute machine code, virtually dictated that only a specific class of programs could be successfully run on early machines. Such were problems that lent themselves directly to simple iterative programs which required only primitive computational resources. When it had become feasible to increase the scope of applications for computers, software tools were constructed to reduce development time. Among these were the mnemoic assembly language and symbolic assembler. These tools helped to reduce the complexity of programming in absolute machine code so that the programmer could better direct his attention to fitting his program

within the hardware dependent constraints of memory space and execution time.

Manufacturers began producing differing machine architectures with differing instruction sets. This diversity made the sharing of software products and tools increasingly difficult and in turn enhanced the desirability for protability in software. Portability would allow the exchange and reuse of already constructed and time-tested routines. Universal desires for portable software and the individual desire of programmers to be able to program in a better or more natural notation than assembly mnemoics led to the first true high level programming language.

While the new language attributes of easier writing, reading and debugging were desirable, inefficiencies in code generation delayed their general acceptance. Computer hardware at this time was still the most costly part of computing. Although technological advancements had been made in bulk storage and in computer size and speed, the computing cost increases due to inefficient compiler generated machine code could not be easily tolerated.

Improvements, gradually provided by technology, increased the speed and capacity of computers to a point which required some method of automatic job sumittal and integration of tasks inside the computer for efficient operation. The operating system concept was a direct result of these new system automation requirements.

Summarizing, this first era of software was characterized by the development and refinement of primitive tools within an environment of time and space containment shaped by relatively high hardware costs.

The second era of software development is characterized by hardware technological growth of exponential proportions. Technological improvements rapidly reduced the costs of computing hardware. This factor and the new sophistication of the operating system allowed a rapid branching of computer applications into many new fields. As users became acquainted with the benefits of computers, their demands in program application, sophistication and size increased rapidly. The effects of hardware cost reduction and user demands caused a revolution in the relative costs of hardware and software. Software costs, which once had been a relatively insignificant part of computing costs, quickly outpriced hardware costs and became a principal factor in new applications for computers. Some current estimates have shown that in the United States between fifteen and twenty-five billion dollars is spent on software annually. Costs to the Department of Defense alone were placed at about three billion dollars annually [4]. Predictions on future costs of software have indicated by 1985 over ninety percent of computing costs may be attributed to software [5]. Meanwhile, hardware technology had provided improvements which, for most applications, removed old constraints of time and space. New constraints came from the life cycle costs of software, the

costs of development and maintenance. Of these, software maintenance has proved the most expensive. The SAGE military defense system, as one example, had a yearly average maintenance cost of twenty million dollars per year after ten years and a corresponding initial development cost of two hundred and fifty million dollars [6]. The Air Force, in another example, reported avionics software development costs of about seventy-five dollars per instruction in 1973 and a maintenance cost of up to four thousand dollars per instruction [5]. A commercial company estimated that eight percent of its total software life cycle cost was also spent in program maintenance [4]. Similar experiences by others have, in most cases, brought goals of software reliability and maintainability from secondary importance into a position ahead of time and space efficiency. As a further cost reducing measure, ease of modification allowing future growth or alteration of a software product has become an important related goal. Characterizing this second era of electronic computing has been the radical change in the relative costs of computer software and hardware. The desire to reduce the software costs has directed much attention to emergent software engineering methods.

B. MODERN SOFTWARE DESIGN PRACTICES

Reliability has been of high importance in modern software practices. Myers [6] defines reliability as:

> "the probability that the software will execute for a particular period of time without a failure, weighted by the cost to the user of each failure encountered." [6, p. 7]

Failures in software are not caused, except perhaps for obsolescence, by "wearing out". Rather, they stem from an error or imperfection in the basic design, the resulting program, or some modification that has been made on the program in the course of its life cycle. Myers [6] indicated that the single major source of these errors was in the process of translation of information. These errors occur in all stages of development and maintenance during a software life cycle. Error producing "noise" can be found in all intra-project communications from the users first proposal to the finished product. The degree of communication that is required in any project phase depends upon the type of project and the type of organization used in its development and maintenance. An example of the effects of noise in a situation requiring a high degree of communication can be seen in a child's game. The game is played by whispering a message from individual to individual. Usually, after only a few of these exchanges, the original message has been hopelessly garbled by "noise". Logically, to make software products more reliable, the extent of communications should be optimally held to a minimum.

Another factor which has been related to the incidence of errors in software is complexity. Fundamental to the human effort of dealing with complexity is the limited

capabilities of the human brain. Studies have shown that an individual can readily assimilate about five to seven independent pieces of information at one time. Problems which go beyond this level of complexity must usually be partitioned into pieces or "chunks". The partitioned chunks represent the consolidation of many smaller pieces into a conceptual group. The group is then given a name that can be used to reference it. This procedure then allows the problem solver to consider a complex system in stages of detail. Each stage, if partitioned correctly, will be within the conceptual limits of the designer [7,8]. If a large problem is not partitioned in this manner or is beyond human preceptual limits, the designer must either resort to estimating with inherent error or enumeration with an accompanying time penalty [2]. In the software life cycle, errors may result from the complexity of the problem, the complexity of the solution and complexity of communications.

In response to the dramatic software cost increases, various new methods and approaches in software development have surfaced. These methods and approaches are meant to reduce costs by promoting software reliability, maintainability and modifiability. Frequently used practices are: structured programming, top-down development, chief programmer teams, HIPO (Hierarchy/Input-Process-Output) documentation, support library and the structured walkthrough. Application of these and related methods or practices in software has

been termed "software engineering". The word engineering implies development through stages of planning, specification, design, documentation, fabrication and testing [4]. Some of these practices are directed at providing a program development environment which is conducive to reliable products: the support library and structured walkthrough are examples. Other practices are directed towards the principal problems of complexity and communications. Myers [6] lists two ways to combat the problems of complexity: make each component of the system as independent as possible and arrange the system in a hierarchial manner corresponding to levels of understanding. The software engineering embodiment of these two principles has been through the use of hierarchial development techniques, top-down design, modular design and structured programming.

1. Hierarchial Development

Goos [9] describes hierarchial or top-down development as: "...Subdivided into many components; every component solves a subproblem into which the original problem can be split..." [9,p.29]. This partitioning of the problem is formed by using a complexity reducing technique called levels of abstraction [8]. Each particular level of system definition is envisioned by the developers as a "black box" or series of black boxes which perform a function or functions. The black box becomes an abstraction of the system at that level of detail. As the development proceeds, each black box

is in turn looked at individually and further divided into a new series of boxes that describe the function in more detail. The process continues until the desired level of definition is reached. The concept of abstraction of processes is fundamental to many software engineering techniques. Similarly, abstraction is the grouping of objects of processes by some common element. The group may then be given a name and thought of collectively without considering many details; a forest is an abstraction for a group of trees, a tree for a group of leaves, branches and a trunk, etc. The concept of abstraction may be advanced to any level of detail. This process can help to make an extremely complex system manageable and yet stay within the psychological capabilities of the developers [8].

2. Top-Down Design

Top-down design was the extension of top-down development to the program design level. Aron [2] described the process as: "... the natural way to design a program unit" [2,p.96]. Levels of abstraction has also been fundamental to this technique. During top-down design the program unit is abstracted to the level at which it should appear in final form; the desired program outputs are established and input requirements are made. The interior of the process skeleton is then filled with the necessary data representations and abstractions of subprocesses which will transform the input data into the output data. This first level of abstraction has then formed a conceptual picture of the

overall structure of the program unit. When design at this level has been completed satisfactorily, each subprocess can then be designed in a similar manner. If a design flaw is found, the process is backed up to the level containing the flaw and redesign takes place. The changes are then passed through all subsequent refinements.

3. Modular Design

Modular design is an extension of top-down techniques. The design process acts to contain system subprocesses in "box like" modules. Besides the benefits of providing an abstraction mechanism to congregate system components, modules can also help control program inter-communication problems by limiting unexpected side effects. In the process of modular design system subcomponents are first defined, then "boxes" are built around them. This isolation can be achieved by designing the modules in such a manner as to have them behave in exactly the same manner each time they are used. Use of a module must require only knowledge of the module interface and not the internal structure [10]. It has been found that this method of design, in addition to being conceptually less complex, is also amenable to design changes. Design changes can often be restricted to the change of a single module [2]. When module interfaces have been made explicit using this technique, inter-communications between elements can be more carefully controlled than in other design methods.

4. Structured Programming

Many different definitions of the term "structured programming" can be found. Each however, follows basically the same vein. Liskov [11] provided this definition:

> "Structured Programming is a programming discipline which was introduced with reliability in mind...Structured programming is defined by two rules. The first rule states that structured programs are developed top down, in levels... The second rule defines which control structures may be used in structured programs. Only the following structures are permitted: concatenation, selection of the next statement based on the testing of a condition, and iteration. Connection of two statements by a goto is not permitted." [11,p.193]

Dykstra [12], in his famous letter which has been said to have fathered the modern concepts of structured programming, expresses the desirability of making the static program or program text represent as nearly as possible the actual process the program was to perform. The thrust here was to reduce the complexity of identifying program text which was related to a specific process action. The more complex this association is, the harder it is to debug, correct errors or modify software. Use of these techniques have been shown to reduce maintenance costs up to fifty percent [13].

Structured programming rules, when added to other techniques such as text indentation in block structured languages, modular program organization and top-down design, have caused dramatic increases in programmer productivity and corresponding decreases in software errors [14].

These briefly described methods generally constitute the major areas of software engineering. Intuitively they have provided an aid to the production of reliable software and in fact, have provided some measurable results. Boeing Computer Services, for example, showed an average weighted improvement of seventy-three percent in software cost in a three-project study [4].

C. SUPPORTING SOFTWARE DESIGN

The effects of top-down development, top-down design and modular design methods have been to reduce an initially complex problem into small manageable pieces called modules. Following this initial design phase, a programmer or a group of programmers is required to translate the design concepts again, using a programming language. The programming language serves two purposes in this step: first, it serves to put concepts into a form which can be machine-translated into a final set of machine instructions, and second, it provides a humanly readable record of the manner in which the final process is to be performed. This record may be read at some later date by others to correct errors. Unfortunately, the nature of present programming languages have caused them to reintroduce complexity into software products. The fundamental ideas behind top-down development and top-down design receive little support from modern high level language constructs. Instead, the programmer must devise ways, using the available features of the chosen

programming language, to implement the design. In doing so some integrity of the design may be lost and new complexity added to the problem. Abstraction, as an example, can be implemented using programming language elements "procedure" and "function". Modularization of abstract processes by procedures or functions is difficult. In most high level languages it is possible to produce unknown side effects with these language constructs. These side effects can influence program operations far removed from their actual location. When procedures or functions are used as a part of a system, communication of data from one part of the system to another may require the use of global data or very complex communication paths. Programming to eliminate side effects and the use of global data in a program causes additional program complexity and is a source of errors. These errors are not part of the original design, but are a result of the inability of the programming language to allow a single translation of design criteria into program text. Structured programming concepts have been developed mainly because programming languages by themselves do not lead naturally to programs which are easily read and contain few, if any, errors. Abrahams [15] noted several difficulties within present-day languages which affect the programmers ability to write structured programs: inadequate linguistic facilities for "goto-less" programming, function and procedure implementation inefficiencies, and language elements which do not naturally produce visual "structure" in a

program. Complexity introduced in a program because of the inability of the language to allow direct implementation of modern design techniques can be further compounded by the complexity of the language itself. Richard and Ledgard [1] noted the unrelenting tendency of modern languages to grow in size and complexity in an effort to satisfy user requirements. The results have been languages with a high level of syntactic and semantic complexity. Subtleties and duplicate forms of expression within a particular language cause confusion and errors. Most high level languages require extensive documentation which is not part of the executed program (program comments, flow charts, logic manuals, etc.) to allow another reader to follow program logic and data flow during program maintenance or modification. Although design procedures are not necessarily restricted to a particular class of problems, the majority of languages are usually best suited to the solution of a limited set of problems. When generality is desired by the user of a specific language, it has usually been implemented by addition rather than redesign. When these additions cause a language to grow to such a size that users frequently require only a small portion of the available language features, or when size of the language is too large to implement easily, a common response is to form a new dialect. Sammet [16], in a roster of programming languages for the year 1974-75 listed one hundred and sixty-seven different current high level languages. This widely diversified set of programming

languages has adversely affected portability of software and the maintenance of installed software systems.

These arguments have been advanced to show that modern high level languages do not sufficiently support the ideas and methodology of modern software engineering techniques. The reasons why these languages do not fully facilitate such practices can be partly explained historically: programming languages were initially conceived and then evolved in an atmosphere of high hardware costs. These costs required programmers to have machine knowledge and program towards efficiency of machine operation, often with a sacrifice in program readability and maintainability. Software engineering practices have been developed to offset the increasing software costs. Historically, most languages have evolved by additions or changes in established bases, future high level languages are likely to suffer from the same problems as their predecessors. Observation of the acquisition process of the Department of Defense language for embedded computer systems (DOD-1) [17,18] tends to support this prediction. The goals and criteria for this new language have been well established and include: reliability, modifiability, efficiency, transportability and generalization. An evaluation of existing languages showed that none satisfied the requirements in an unmodified fashion. Examination of the preliminary reports of the four final competing designs revealed that the requirements had been met by additions

both syntactically and semantically to the base language Pascal with a corresponding increase in language complexity.

D. SUMMARY

The development of programming languages has been a relatively slow and evolutionary process as compared with the development of software engineering methods. Memory space and execution time efficiency due to significant hardware costs were the main driving forces of early programming language design. In recent history, technology has caused hardware costs to drop dramatically in relation to life cycle costs of software. With the high cost of software development came the realization the design methods could be applied to software construction and could cause a positive impact in lowering overall cost. This impact was in part due to reducing problem and software system complexity and accompanying errors as well as facilitating program maintenance and modifications. Rather than integrating languages into these design methods, the methods were applied to languages. The high level languages were further molded toward conformance to these engineering techniques by the application of structured programming rules.

III. DESIGN APPROACH

Two philosophical approaches were evaluated as a basis for the design of a programming language which would be better suited to the production of software in a modern software engineering environment. One of these approaches was to use an already existing language as the design basis. A strategy in this instance was to examine current high level languages for the most suitable example. After selection, the structure of this language would be modified in the areas where it was felt that weaknesses were present or significant gains could be made. This particular approach was used in acquisition procedures for a new language for the Department of Defense (DOD-1) [17]. An examination of languages built on this approach revealed a tendency to become much more complex than the original base language. This relative increase in complexity was observed to be in part due to appending the desired features rather than integrating them into the host language. This first method, using a host language for the base, was discarded as impractical because it was felt that a complete redesign would be required for any existing host language to meet design criteria.

The second alternative was to design a language without relying on an existing language as the host. A strategy in this case was to base a language design specifically on a

set of guideline principles, that of software engineering practices, and information gained by observing some relevant areas of problem solving and human behavior. In this manner, the design would be based on fundamental design procedures and would emulate the way that human beings attack the solution of problems and the construction of systems. This second alternative was selected because it was felt that such an approach would be most likely to produce desirable results.

A. OBSERVATIONS IN PROBLEM SOLVING AND HUMAN BEHAVIOR

Simplicity of use and the integration of top-down design concepts were considered as the primary goals in this language design. Top-down techniques in design and program construction are fundamental aspects of the concept of abstraction. Abstraction is an observable complexity reducing mechanism which is used naturally by human beings in the problem solving process. Such abstraction, the collection and categorization of thoughts, becomes a necessity when the complexity, that is, the number of elementary problem components, becomes greater than the individuals' intellectual capacity to deal with them. The process of abstraction is also a convenient mechanism to avoid rethinking each individual detail or step of an object or activity which has been "learned" from previous performance or experience. When an individual is asked to "walk to the door", for example, the activity of walking is usually not thought of as a

number of specific detailed movements. The individual merely applies a previously learned abstraction of the activity of walking. A specific instance of this abstraction process, in this case "walking", may require reference to additional data, i.e., "to the door". Reference to a particular element of a class allows an abstraction of a process or object to be generalized and applied to all elements in this class. When learned as such a generalized process or object, application to produce a specific result is done simply by referring to the abstraction by name and supplying the missing information. In this manner, application of a generalized abstraction has saved the time and possible errors which would be introduced if the process was rethought completely step by step each time it was accomplished.

Relatively simple abstract processes may act in a cumulative fashion to yield relatively more complex objects or processes. Details previously abstracted and remembered may be combined with others to perform an action more complex or solve a problem beyond the capabilities of the individual contributing elements. During the collective application of these individual processes, information could be produced which may be relatively unimportant when the final product is viewed, but was essential in determining proper courses of action or the handling of data between each individual elementary abstract process. Once the final process achieves the desired results, it may be absorbed into a

larger encompassing abstraction. At this point all interior processes and information become relatively unimportant when viewed at the level of the encompassing abstraction. Simplistically, there are two "memory" processes illustrated here. The first is a short-term memory process. This memory process supplies temporary facilities for storage and retrieval of information which is developed in or passed among element abstractions within an encompassing cumulative type abstraction. Continuity within such encompassing abstractions is maintained by short-term memory. Referring to the previous example, "walking" is an encompassing abstraction. It encompasses smaller elemental abstractions such as "raise left leg". When viewing the process from the level of elementary abstractions, continuity via short-term memory is important. It is important, for instance, to be able to recall from short-term memory whether the right leg is still up from a previous use of that process when the "left leg up" process is about to commence. Details such as this are relatively unimportant when viewed from the level of walking. The second memory process is a long-term memory process. This memory action allows learning type actions. Learning activity involves moving an abstraction into storage for use in some future application which may be unknown at the time of storage. In long-term memory, an abstraction may be retrieved for individual use or to become part of a cumulative type abstraction.

Abstractions can serve as tools of communication between people who have learned the same abstractions. Use of special terms and emphasis on a set of distinct descriptive phrases are some of the ways that people within a specialized area communicate. Such special terms and phrases form highly efficient abstractions which convey information accurately from one informed individual to another. Although highly effective when used between individuals who are both familiar with the abstraction, these terms may be completely ineffective when attempting communications between individuals of different backgrounds. This situation exists in the development of software systems. The originator and eventual user of the system must rely upon computer specialists, often not knowledgeable in the user's field to produce the software product. Misunderstanding caused by the inability to effectively communicate across the user-computer specialist interface is a source of unfulfilled requirements.

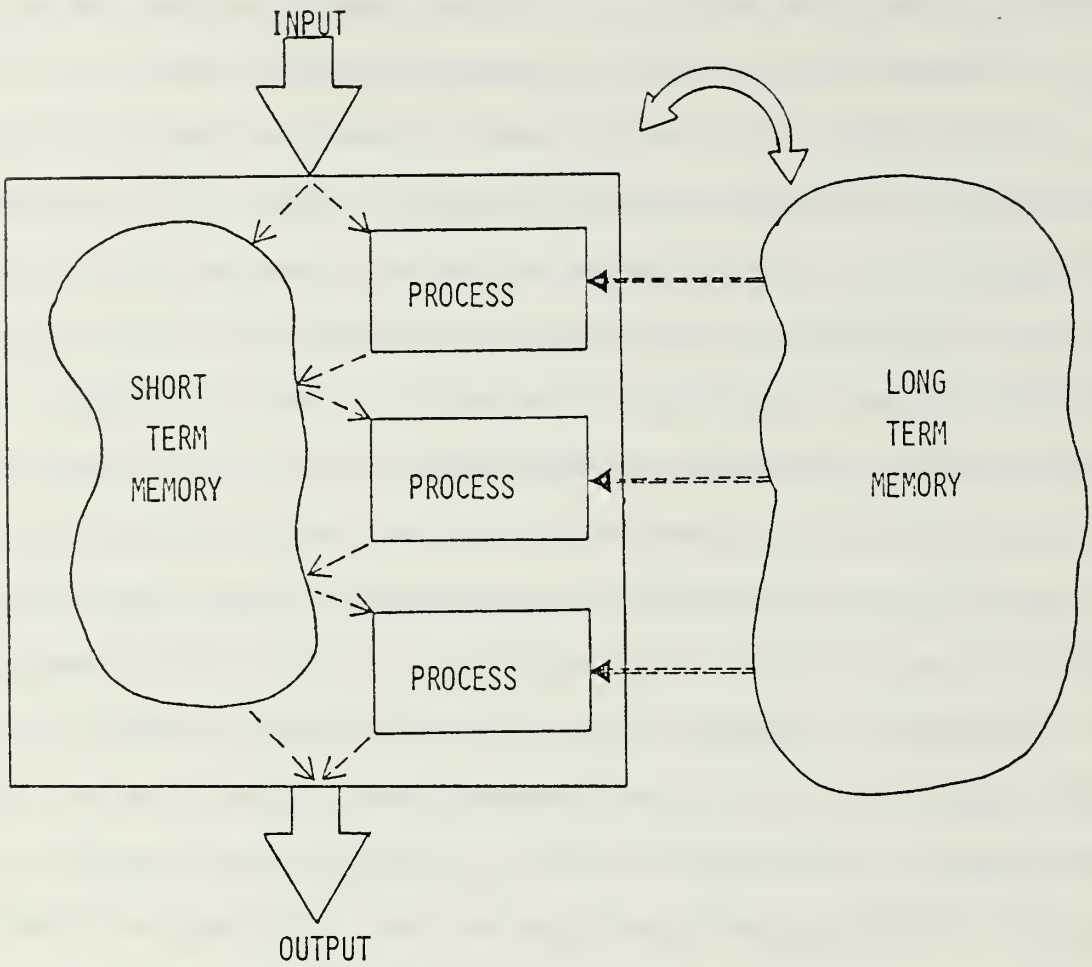


Figure 3-1

B. SYSTEM MODEL

Figure 3-1 represents a system model in the manner in which abstraction is used in everyday activities. An action abstraction, such as "walking", is a self-contained generalized process (figure 3-2A). The process utilizes a set of input parameters, "when", "where", "how much", etc., and produces both an appropriate action and some residual outputs. The residual outputs may be applied to a following connected process. A residual output for a walking process

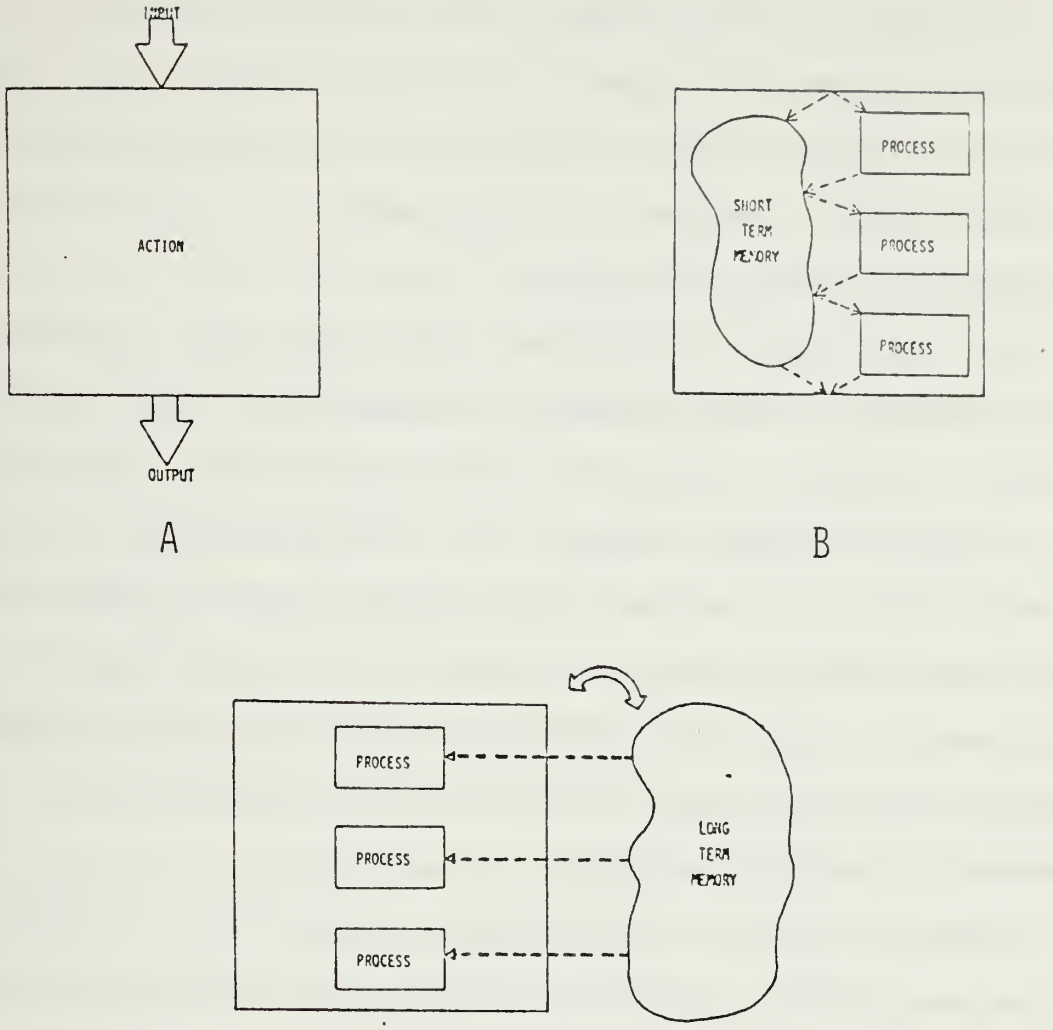


Figure 3-2

might be the signal "I've arrived at my destination and I'm ready for the next instruction". The action portion of a process, although conceptually singular, actually embodies several distinct subprocesses (Figure 3-2B). Each of these separate more elementary processes, moving the right leg up, for example, are connected together in some logical sequence

to produce the desired effect. Continuity is provided between subprocesses by the use of short-term memory. This memory is used to retain information that is important to the subprocesses, but not important enough to retain upon completion of the major action.

Once a process is correctly constructed from its subprocess elements, or an object is constructed from its subelements, it may be "learned" in entirety by being moved into long-term memory (Figure 3-2C). This memory is a resource from which action abstractions or object abstractions may be drawn when required for future applications. Additionally, entire objects or processes may be reconstructed from elementary parts if these parts had likewise been retained in long-term memory.

The entire system of processes (or objects) and memory form an environment in which abstractions can evolve through creation and refinement to fit specialized requirements. Specialized object and action tools for direct user communication may be built in such an environment. Improved communications at this level may lead to increased software reliability and greater fulfillment of user requirements.

In addition to supporting observations of everyday human uses of abstraction, this model also supports the concepts and practices of abstraction in top-down design. It was felt that this model was a reasonable basis for the design of a programming language which would be more capable of supporting modern software design techniques. A language constructed

on such an underlying basis would provide many benefits. One of these benefits would be the ability to apply abstraction and top-down techniques uniformly from design conception down to final program details. Because the procedures would remain fixed throughout the design process and proper communication aids in the form of abstractions could be provided, the user would be able to better participate in the specification and eventual construction of his system. An additional important benefit is that of machine independence. Because abstractions can be refined to an absolute base of primitive abstractions, portability in software may be achieved easily by supplying this base in a new machine. Simplicity and uniformity provided by this approach to software construction should decrease program construction time, debugging time and eventual modification or maintenance efforts. Language "learning" capabilities in the form of a library of abstractions could allow greater individuality in a language without increasing the size of the language.

C. LANGUAGE CONSTRUCTS

In an effort to further localize specific areas requiring special attention in a language design, language constructs found in the modern programming language PASCAL and preliminary DOD-1 proposals were examined. This examination was made to reveal potentially troublesome constructs for users of the language. In this manner, it was hoped that areas of complexity, non-uniformity, and little-used features would

then receive special attention in the new language design. In the course of this examination, the following areas were identified: if-then-else constructs, global variables, data typing and extension facilities.

1. If-Then-Else Constructs

It was felt that if-then-else structures, when nested, caused difficulties in the determination of which parts of the nested structure belonged together. It was also observed that this structure, although relatively easy to write initially, was hard to modify easily.

2. Global Variables

Existence of global variables causes increased complexity in the sense that these variables provide more possibilities for the propagation of program side effects. Since these side effects may produce errors in locations far removed from the initial effect, debugging time may be increased substantially.

3. Data Typing

It was observed that data typing, the categorization of data by its type, made programs potentially hard to read. This reading difficulty was due to the degree of mental "threading" that was required to locate all the information with data declarations that pertained to a specific variable name.

4. Extension Facilities

The languages which were reviewed either lacked easy facilities or had no facilities to extend the language

base by installing user defined features within the system. Because extension facilities were limited, the languages exhibited the characteristic of having desired features appended to rather than integrated into the language base. The overall effect was an increase in language size and complexity. Since all features were directed towards specific language applications, users would tend to use a subset of the language; the remainder of the language would be relatively unused, but carried by the translator to complete specified language requirements.

D. DEFINING THE LANGUAGE

The language definition found in the accompanying base language report was produced in a top-down manner from the abstract system model described earlier. Abstractions which emulated this model structure were first designed for data and processes. Tools for data and control structures in processes were next designed with specific goals of uniformity, conceptual simplicity and strict adherence to a fundamental policy of containment of data within a program process element. Once all major structures were defined, refinements were made to optimally achieve all goals set for the final design.

The vehicle chosen to produce this language design was the syntax diagram. It was felt that a definition based on diagrams afforded the reader the best picture of both language syntax and structure.

IV. BASE LANGUAGE REPORT

A. INTRODUCTION

An attempt has been made to develop a computer language which is relevant to many applications, yet has a small, simply stated grammar which is easy to learn and easy to use. These goals cannot be met with a language which incorporates into the syntax all probable application requirements such as PL/I and the proposed DOD-1 languages. Application requirements are not static, they change with time. A language which incorporates capabilities into the syntax or grammar cannot be expected to keep up with dynamic application requirements. Either the language or the application must change so that the two are compatible and usually it is the application which is molded to fit the computer language.

The approach taken, as described in the previous section, to meet the goals of developing a small, simple language which could be used in many applications was to provide a limited syntactic base. From this base, application oriented processes could be constructed and saved for future use in a language translator controlled system's library. The syntactic base which was developed is called the base language. It provides general capabilities which are necessary for all computer languages such as execution control structures, expression evaluation and data value assignment. In addition, it provides the capability to define and save for future use,

application oriented data structures and processes. The user defined application processes and data structures used in conjunction with the base language form a language which is uniquely tailored to the particular application. A language which the user may alter to meet changing applications.

The base language report describes the grammar and use of the base language. Liberal use is made of syntax diagrams, schematic representations, and examples of base language code in an attempt to simply and clearly present the language. The examples are presented in upper case characters for clarity and do not imply an implementation which is restricted to upper case. A top-down approach is taken in presenting the base language. First, a brief overview of the language structure is presented. This is followed by a detailed description of the grammar. Each section of the description is presented in a top-down manner which describes the principal component followed by descriptions of the next level of components and so on until the most primitive elements are described. Following the base language description some proposed basic data structures and processes are presented.

In the description of the base language a conscious attempt was made to present only the syntax, semantics and concepts of the base language and when possible not to influence the implementation of the base language translator.

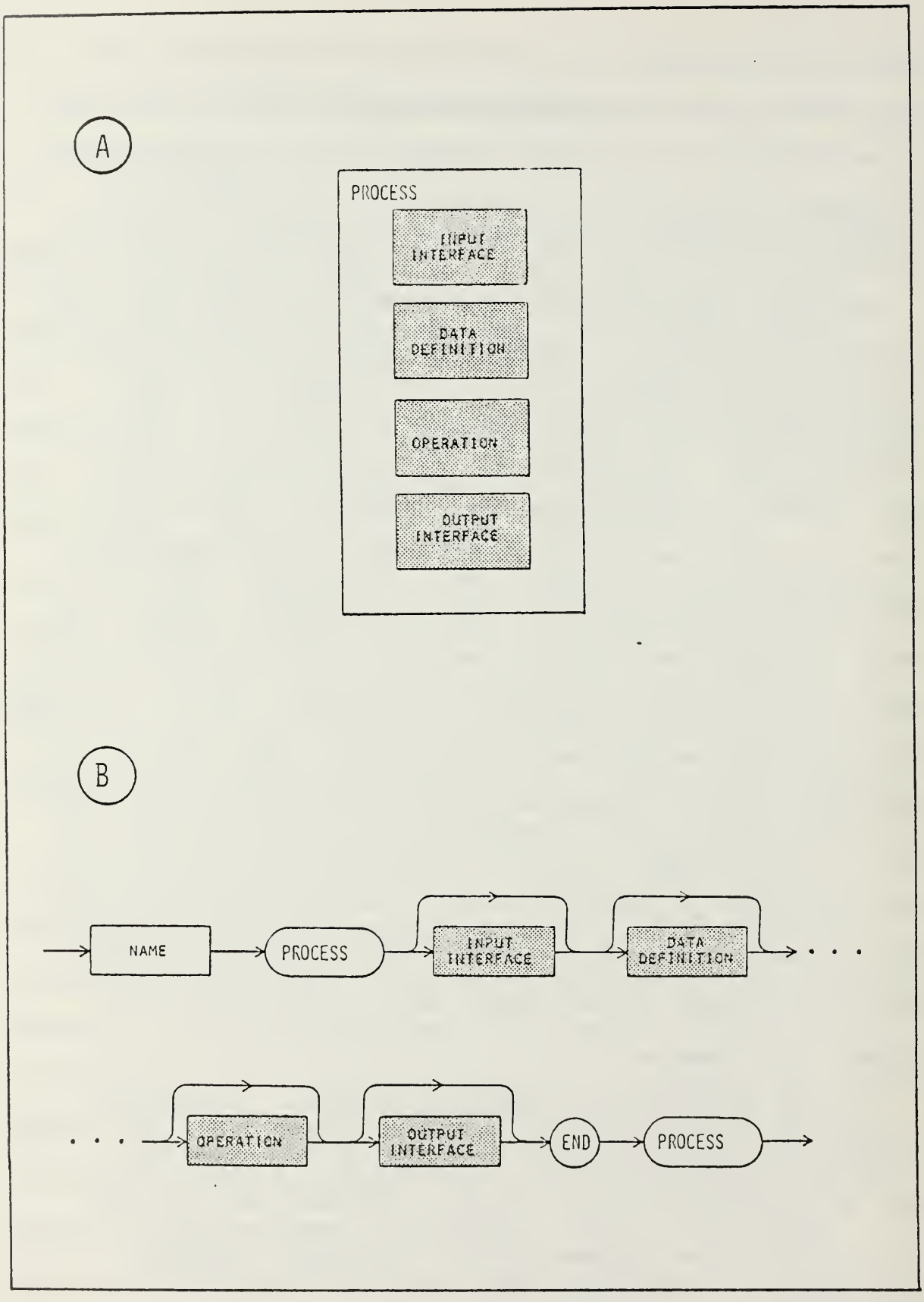

A
PROCESS
INPUT INTERFACE
DATA DEFINITION
OPERATION
OUTPUT INTERFACE
B
NAME
PROCESS
INPUT INTERFACE
DATA DEFINITION
OPERATION
OUTPUT INTERFACE
END
PROCESS

Figure 4-1

Example IV - 1: Computation of the greatest common divisor

```
GCD PROCESS
    INPUT
        X, Y
    END INPUT
    DATA
        X, Y: IMPORTED D;
        A, B: INTEGER
    END DATA
    OPERATION
        A←X; B←Y;
        REPEAT ((A-B): TERMINATE)
        {

            SELECT
            {

                (A>B): A←A-B;
                (DEFAULT): B←B-A;
            };

        };
    END OPERATION
    OUTPUT
       A
    END OUTPUT
    END PROCESS
```

B. BASE LANGUAGE OVERVIEW

1. Process Structure Overview

The process is the principal component of this language. It is analogous to a "program" in many traditional computer languages. Processes are the units of the base language which perform actions. A process is a self contained executable module which can be constructed to perform a task unsupported by other processes or as a "sub" unit which relies on other processes for information. A process is composed of four sections as shown in Figure 4-1. First, the input interface section provides a single controlled

entry point for data used by a process. Second, the data definition section defines all data names which are used in the process. Third, the operation section forms the active portion of a process which transforms the data in the desired manner. Last, the output interface section provides a channel for passage of information to other processes. The process declaration provides a means of naming a process so that the process can be used as an abstraction which is referenced by name.

Example IV - 1 shows the form of each of the sections which make up a process. The indentation shown is not required by the syntax of the language but is recommended and when used makes the structure of the process more apparent. Each section of this example will be discussed in the following sections of the base language overview.

THIS PAGE INTENTIONALLY LEFT BLANK

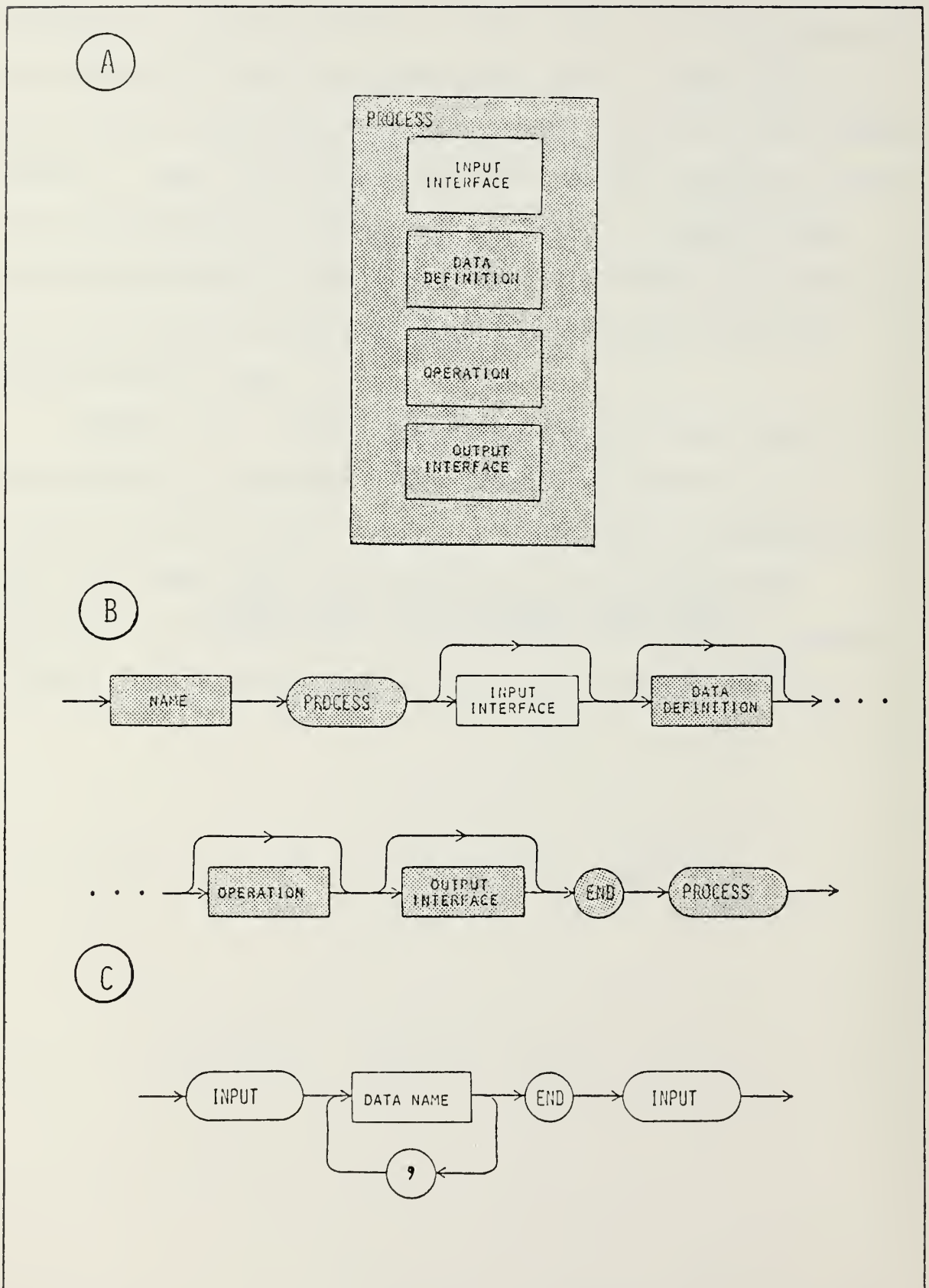

A
PROCESS
INPUT INTERFACE
DATA DEFINITION
OPERATION
OUTPUT INTERFACE
B
NAME
PROCESS
INPUT INTERFACE
DATA DEFINITION
OPERATION
OUTPUT INTERFACE
END
PROCESS
C
INPUT
DATA NAME
END
INPUT
,

Figure 4-2

Example IV - 2: Input Interface Section from the GCD process

```
INPUT
    X,Y
END INPUT
```

2. Process Interfaces Overview

a. Input Interface Overview

The input interface is the first section of the process. It defines an input channel connection for data being passed into the process by associating a process data name to each data value passed.

The input interface is constructed by listing process local names corresponding to data items external to the process which are required for performance of the process task. This interface is the only point at which external data values may enter a process.

The structure of the input interface is shown in Figure 4-2.C and in the above example. The input interface is described in detail in Section IV.C.3.b.

Example IV - 2 repeats the input interface section from Example IV - 1. It indicates that the values of X and Y are external to the GCD process and are to be passed in for use by this process. This section shows the reserved words INPUT and END INPUT which delimit the input interface section. Between these delimiters a list of data names, X and Y, make up the formal parameter list of the input interface. The indentation and multiple lines used in this example are not required by the syntax but are

recommended to make the section more recognizable. The following single line input interface section is also syntactically legal:

```
INPUT X, Y END INPUT
```

THIS PAGE INTENTIONALLY LEFT BLANK

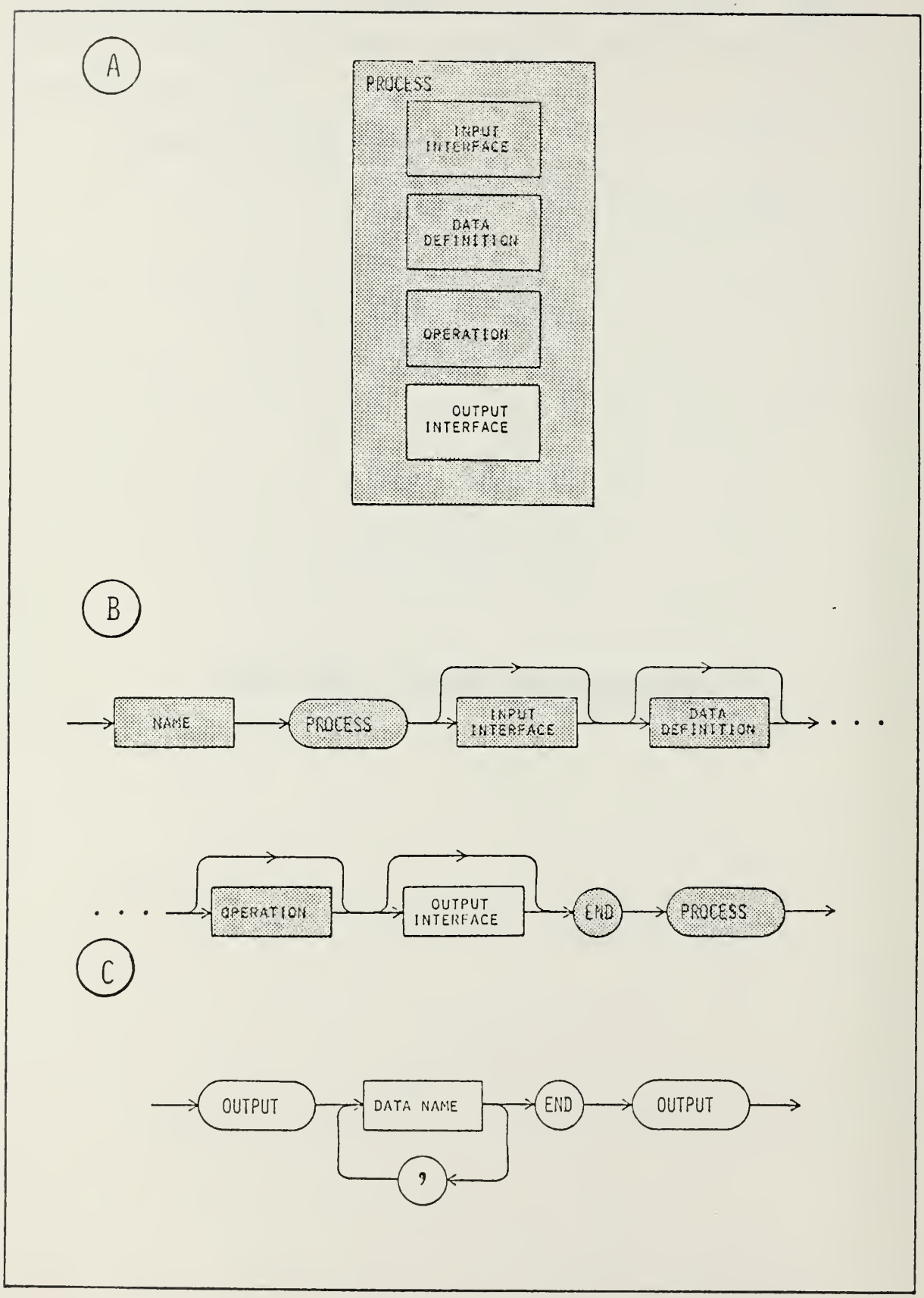

A
PROCESS
INPUT INTERFACE
DATA DEFINITION
OPERATION
OUTPUT INTERFACE
B
NAME
PROCESS
INPUT INTERFACE
DATA DEFINITION
OPERATION
OUTPUT INTERFACE
END
PROCESS
C
OUTPUT
DATA NAME
END
OUTPUT
,

Figure 4-3

Example IV - 3: Output Interface Section from the GCD process

```
OUTPUT
    A
END OUTPUT
```

b. Output Interface Overview

The output interface is the last section of a process. It defines the output channel connection for data being passed out from a process and the data names associated with the connection. Data passed out via the output interface may be subsequently used by another process as input data.

The output interface lists the names of data items used in a particular process which are to be passed out for subsequent use by other processes. It is the only point at which a process may exit and is the only way that data may be passed out of a process for use by another process.

The structure of the output interface is shown in Figure 4-3.C and in the above example. The structure is described in detail in Section IV.C.3.c.

Example IV - 3 repeats the output interface section of Example IV - 1. It indicates that the value of A, which is computed in the GCD process, is to be passed out to an encompassing process which used the GCD process. The output interface section is delimited by the reserved words OUTPUT and END OUTPUT. The data names whose values are to be passed out of the process are listed between these delimiters.

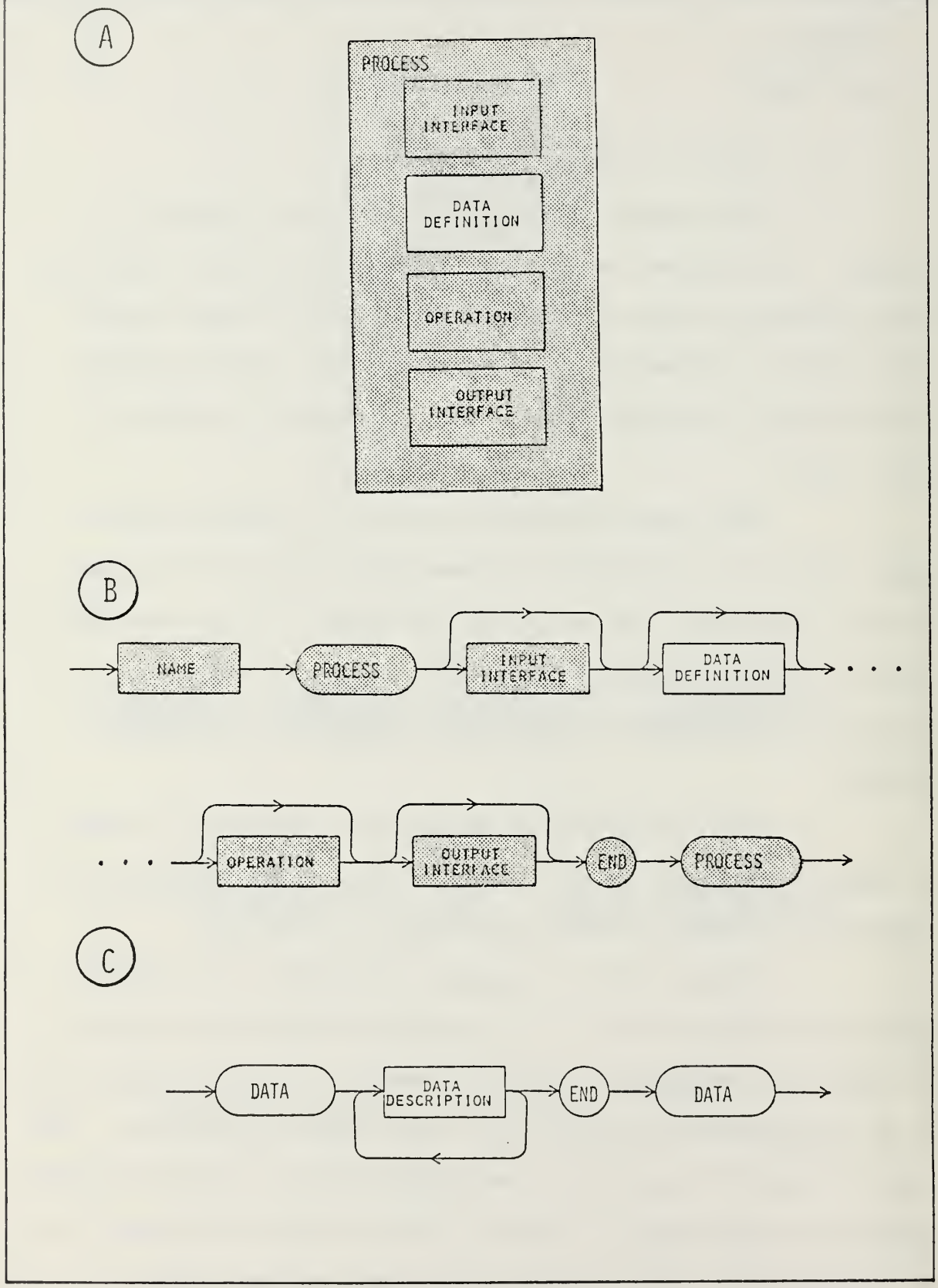

A
PROCESS
INPUT INTERFACE
DATA DEFINITION
OPERATION
OUTPUT INTERFACE
B
NAME
PROCESS
INPUT INTERFACE
DATA DEFINITION
OPERATION
OUTPUT INTERFACE
END
PROCESS
C
DATA
DATA DESCRIPTION
END
DATA

Figure 4-4

Example IV - 4: Data Definition Section from the GCD process

```
DATA
  X,Y: IMPORTED D;
  A,B: INTEGER;
END DATA
```

3. Data Definition Overview

The data definition section is the second section in a process. It normally follows the input interface section This section lists all data names used in the process, describes the structure or property of each and allows amplifying information to be associated with the data item. The property of a data item associates a data structure with the listed data names. The amplifying information associated with a data name provides data values and limits on the value that a data item may take. More detailed information concerning data properties and amplifying information is presented in Section IV.C.4.e.

The structure of the data definition section is shown in Figure 4-4.C and in Example IV - 4. It is composed of a sequence of data descriptions which are described in detail in Section IV.C.4. Example IV - 4 is a copy of the data definition section from Example IV - 1. It indicates that there are four data items being used in the example GCD process. The names of these data items are X, Y, A, and B. The property associated with data names X and Y is the built-in property IMPORTED. This property indicates that the property of this data name has been described in some enclosing process as a data item named "D". The property

of this data item, as a result of the imported property, is to be used unaltered as the X and Y property. A process that uses the IMPORTED property within its data definition section may not be saved in the system library via the LEARN directive. A process may be "learned" only if all properties are fully defined. The property of data names A and B is INTEGER. This property is a user defined data structure. No amplifying information is defined for the data items in the GCD process. The data definition section is delimited with the reserved words DATA and END DATA in a manner similar to the input and output interface sections.

THIS PAGE INTENTIONALLY LEFT BLANK

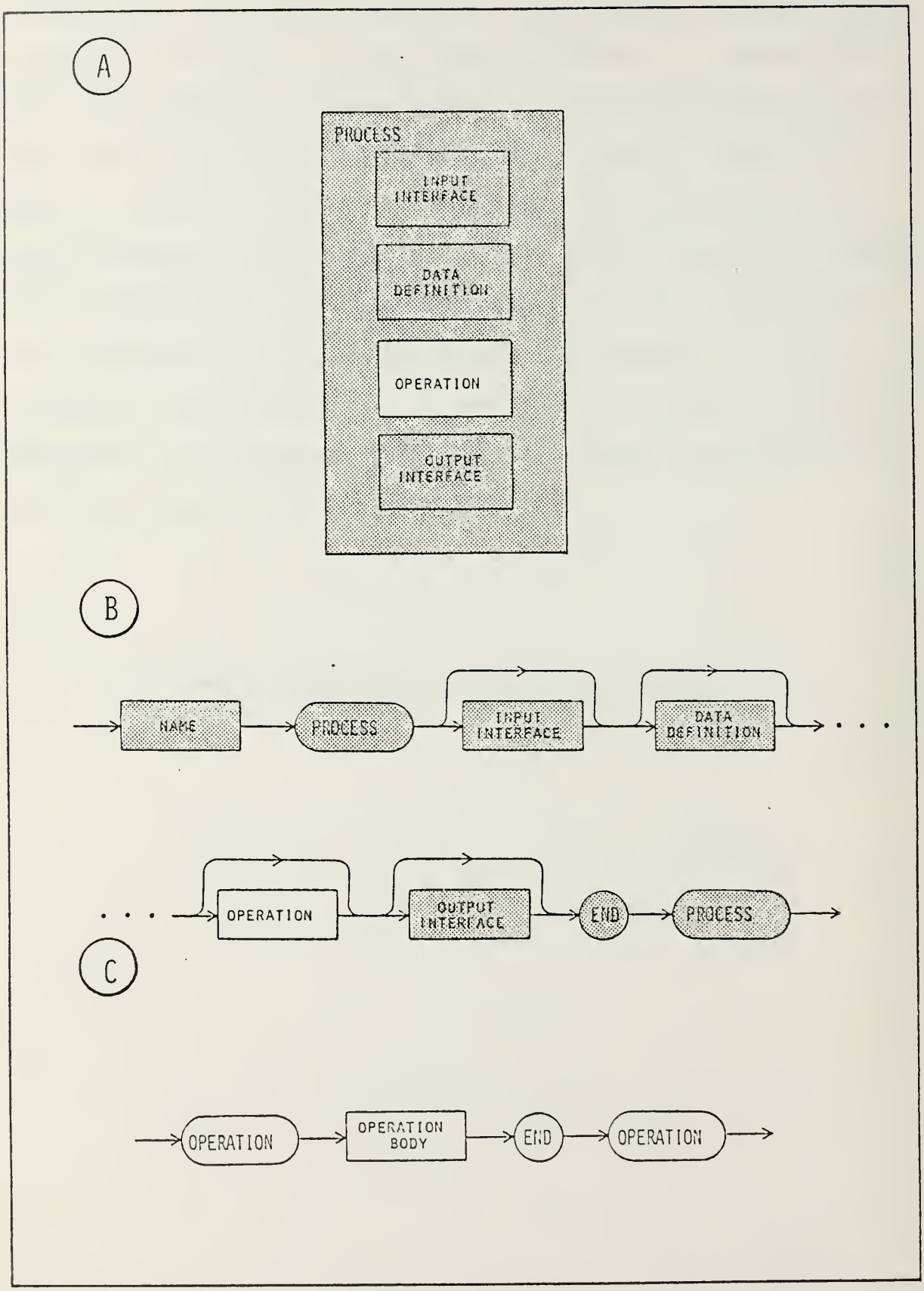

A
PROCESS
INPUT INTERFACE
DATA DEFINITION
OPERATION
OUTPUT INTERFACE
B
NAME
PROCESS
INPUT INTERFACE
DATA DEFINITION
OPERATION
OUTPUT INTERFACE
END
PROCESS
C
OPERATION
OPERATION BODY
END
OPERATION

Figure 4-5

Example IV - 5: Operation section from the GCD process

```
OPERATION
  A←X; B←Y;
  REPEAT ( (A=B): TERMINATE)
  {

    SELECT
    {

      (A>B): A←A-B;
      (DEFAULT): B←B-A;
    };

  };
END OPERATION
```

4. Operations Overview

The operation section is the third section in a process and follows the data definition section. The operation section is the active part of the process. In this section a task is performed or a problem is solved through the use of base language operation elements, or predefined processes which are stored in the system library. The operation section uses names defined in the data definition section to perform the desired task.

The structure of the operation section consists of an operation body enclosed within the reserved word delimiters OPERATION and END OPERATION. The operation section is depicted in Figure 4-5 and Example IV - 5.

Example IV - 5 shows some of the base language operation elements. It should be noted that each operation element is terminated by a semicolon and groups of elements are contained by left and right curly brackets ({ }). A complete discussion of all operation elements is presented in Section IV.C.5.

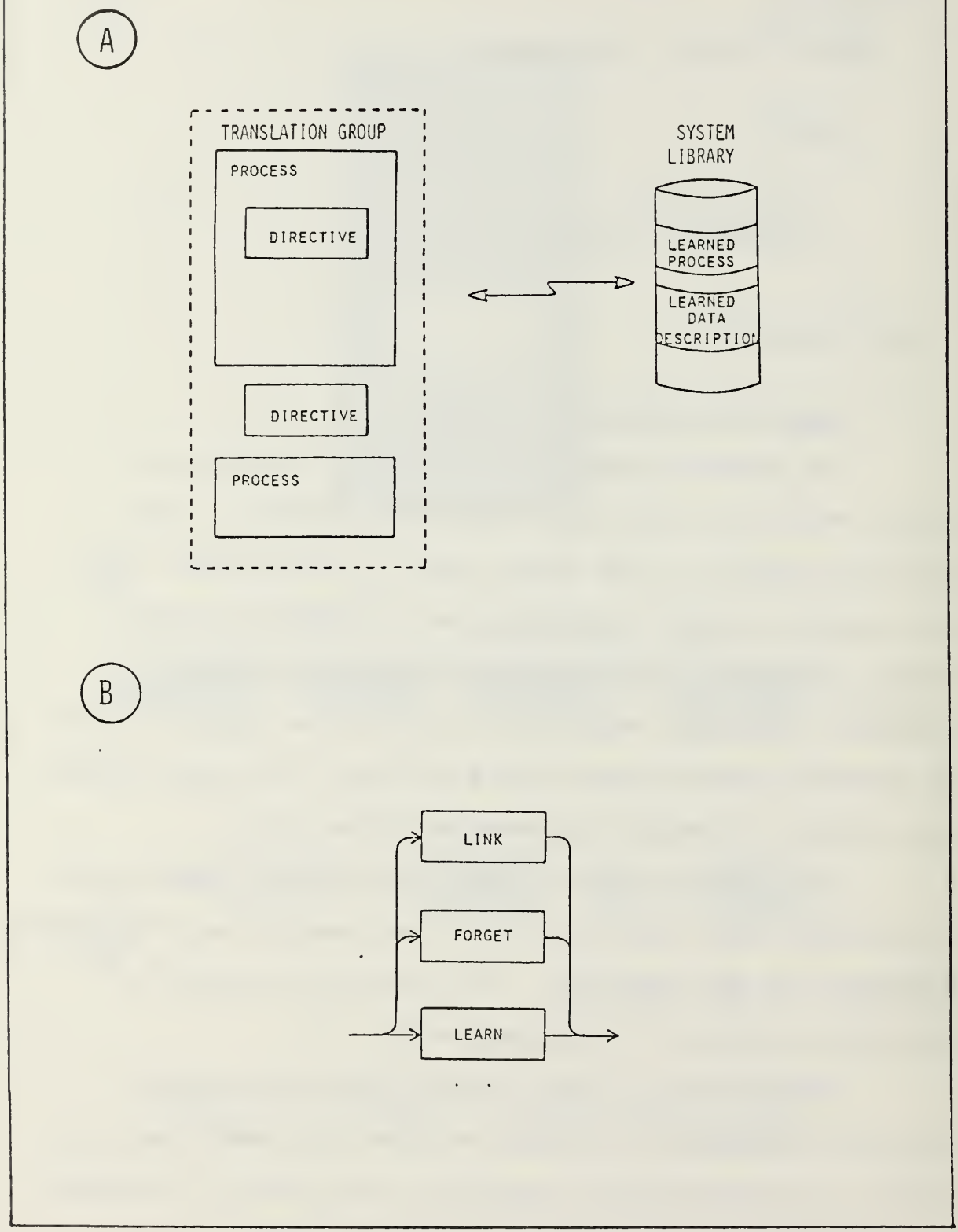

A
TRANSLATION GROUP
PROCESS
DIRECTIVE
DIRECTIVE
PROCESS
SYSTEM
LIBRARY
LEARNED
PROCESS
LEARNED
DATA
DESCRIPTION
B
LINK
FORGET
LEARN

Figure 4-6

Example IV - 6: Sample Learn, Forget and Link Directives

a.
```
!LEARN! GCD PROCESS
  # PROCESS SECTIONS
END PROCESS
```

b.
```
!FORGET! FORM1040;
```

c.
```
!LINK!*:REALMULT, INTMULT, MIXMULT;
```

5. Directives Overview

Directives are commands to the base language translator. They provide a means of communicating with the translator for the purpose of placing a defined process or data structure in the system library, removing processes or data structures from the system library or linking predefined processes to an operation symbol.

When a process or data structure is placed on the system library it is in effect "learned" by the system. A LEARN operation is performed by preceding the process of data structure with a LEARN command (!LEARN!) as shown in Example IV - 6a. Once a process has been learned by the system it can be referred to by any process. The system library is a file system which is created during base language implementation and controlled and updated by the base language translator. A process may be learned only if all properties in its data definition section are fully defined. The use of an IMPORTED property in a process data definition section precludes the "learning" of that process and will generate an error during translation if a LEARN directive is used in conjunction with the process.

Removing processes or data structures from the system library is accomplished by preceding the name of the process or data structure with a FORGET command (!FORGET!) as shown in Example IV - 6b. In this example the translator is directed to remove an item named FORM1040 from the system library. From the name it can be assumed that the item FORM1040 is a data structure but this may not be the case since the type of data item is not stated. This points out the fact that learned items should be uniquely named or tagged by the system to avoid referencing problems or inadvertent removal from the system.

The LINK directive provides the capability to associate one or more processes with an infix arithmetic operation symbol. Once a process is linked to an arithmetic symbol, the symbol may be used in place of the process name in the normal infix manner. The structure and use of the directives is discussed in detail in Section IV.C.6.

C. BASE LANGUAGE DESCRIPTION

1. Notation and Terminology

In this section the base language is described in detail through the use of syntax and lexical diagrams. Each diagram describes the structure of a syntactic or lexical element. By tracing a path through a diagram in the direction of the connecting arrows, the reader can produce an instance of the syntactic or lexical element represented by the diagram. The following two sections describe in detail the rules for the use of syntax and lexical diagrams.

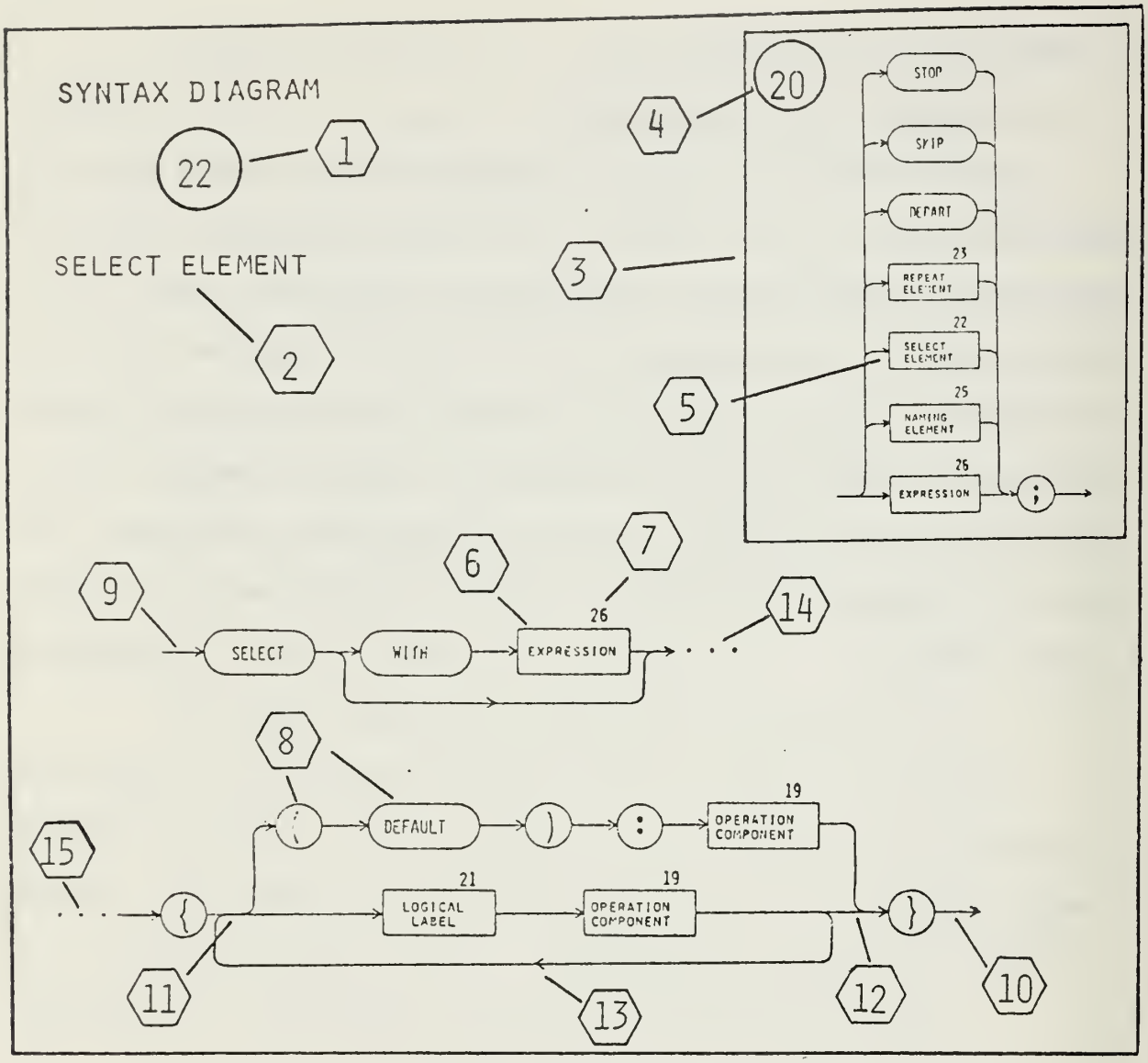


Figure 4-7

a. Syntax Diagram Rules

Syntax diagrams are used in this section to explain the structure of this language design. Each syntax diagram defines the set of legitimate instances for a particular kind of syntactic element. The syntax diagram number, shown at (1) serves as a reference number for a diagram. The name of the syntax element being defined

appears immediately below the syntax diagram number at (2). The syntax element in Figure 4-7 is a select element.

The syntax diagram which is hierarchically the predecessor of the syntax category being currently defined is shown in a rectangle in the upper right corner of the diagram at (3). The predecessor diagram is depicted in an attempt to provide the reader a point of reference and a certain amount of continuity. In a few cases no predecessor syntax diagram is shown. This indicates that in this instance there is no predecessor for the syntax element being defined. The syntax diagram number of the predecessor element is shown at (4) for reference. The syntax element which is being defined is shown in the predecessor diagram (5).

Rectangular boxes such as (6) represent non-terminal syntactic elements. Non-terminal elements are further defined in other syntex diagrams. The number above the box (7) is the syntax diagram number which further defines the element. Rounded boxes or circles, shown at (8), represent syntactic terminals or elements which have no further definition. These syntactic terminals are single characters or sequences of contiguous characters.

To generate an instance of the syntactic element, the reader should follow the diagram, in the direction of the connecting arrows, from box to box. The starting point is always at the left end of the diagram, as shown by (9). The ending point is at the right end of the diagram, as

shown by ⟨10⟩ . In following a diagram, when the reader reaches a junction point ⟨11⟩ any of the paths leaving the junction may be taken. It is not legal to back up along a convergent path ⟨12⟩ . Sometimes, potentially infinite loops, such as ⟨13⟩ may be encountered. The number of times these loops may be traversed is a language implementation dependent limitation.

If a diagram does not fit horizontally on a page it is broken into segments indicated by three dots at the end of one line ⟨14⟩ and three dots at the beginning of the next line ⟨15⟩.

THIS PAGE INTENTIONALLY LEFT BLANK

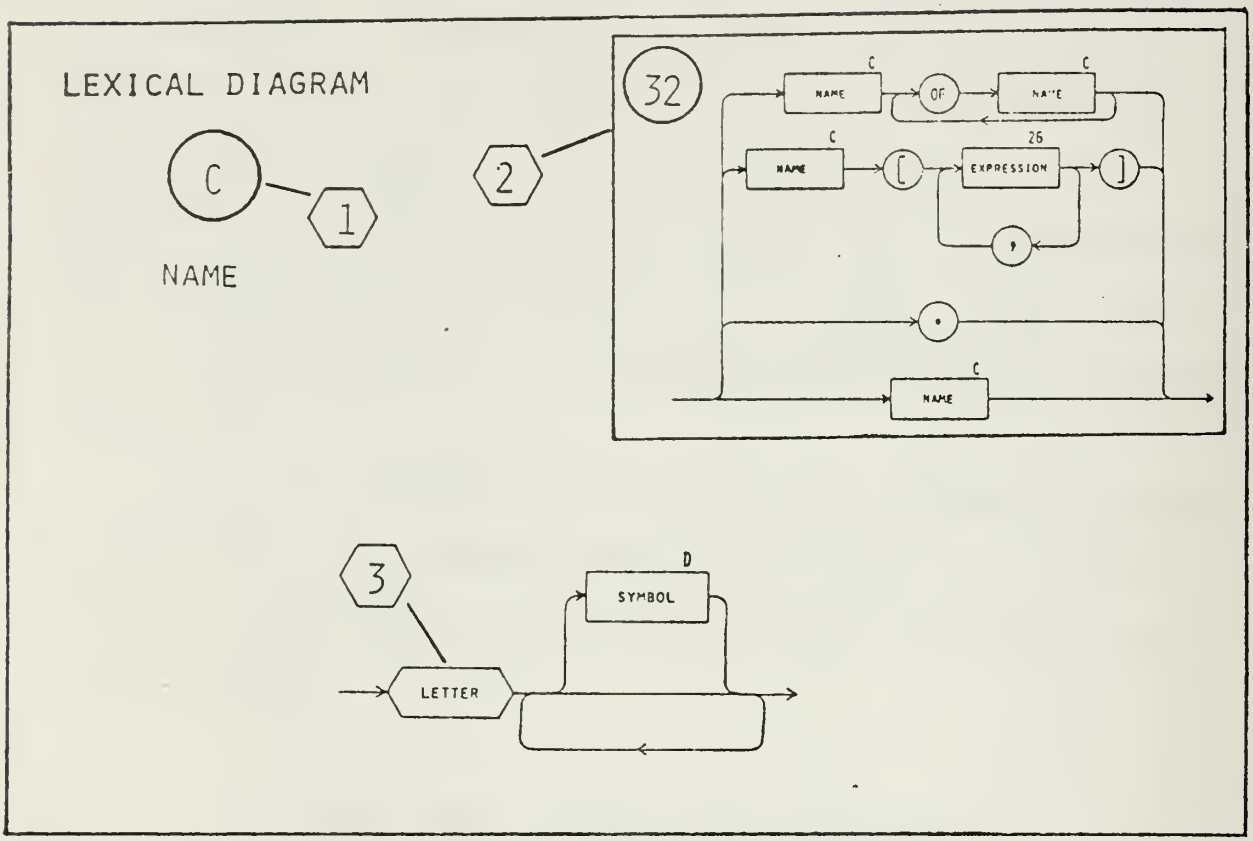


Figure 4-8

b. Lexical Diagram Rules

Lexical diagrams depict the groups of characters in the source process which logically belong together. These groups are called tokens. The rules for lexical diagrams are the same as those for syntax diagrams with the following modifications.

The lexical diagram uses a letter instead of a number as a reference index. This is shown in ⟨1⟩. The predecessor diagram ⟨2⟩ may be either a syntax diagram or a lexical diagram. Boxes with pointed sides, as shown by ⟨3⟩, represent characters from the base language character set. These characters are defined in Section IV.C.7.

c. Terminology

Some of the reserved words used in defining the syntactic structure of the base language are similar to words used in other languages, however, the semantics of many of these words are not the same. The reader is cautioned not to assume the meanings of reserved words based on previous experience with other computer languages. The meanings of all reserved words, their purpose and their use is described in detail in Sections IV.C.2 through IV.C.7.

2. Process Structure

a. Basic Concepts and Scope

The process is the principal component of the base language. It is analogous to the program in other computer languages. A process is a closed, translatable and executable unit which performs a specific function or sequence of functions. One or more processes may be grouped together for translation and execution. Grouping process for translation and execution allows any process declared in the group to reference (call) all processes within the group. In effect, the processes in the group that are referenced are equivalent to processes in the system library (Section IV.C.6.) and would be handled in the same manner by the translator. Unless processes within the group are "learned", they are not retained within the system for later use.

A process accomplishes its function by performing operations on data. There are two primary classes of data in the base language; system data and process data. System data is data which is external to all processes, that is, data which is read into or written out of a process by system input/output processes. Process data is data which is local to the process in which it is defined and external to all other processes.

Figure 4-9 is a schematic of data control within a process and abstractly represents the structure of a process. All processes have a structure identical to that

of process; which consists of input and output interfaces, a data definition section and an operation section. The data definition section provides the information necessary to create a "pool" of data for process ABC. This pool is active only when process ABC is active, that is, data can be used from this pool only by process ABC and only when process ABC is being executed. Data integrity is achieved by this mechanism. The operation section, the active component of a process, may perform operations only on process data defined within the process. For example, the operation section of process ABC can perform operations on data associated with the names ARF, L, M, A, B and C. The input and output interfaces provide the only means of transferring process data from one process to another, thus changing the local environment of the data.

Example IV - 7 provides a sample process which corresponds to Figure 4-9 and can be used to show data control within a process. Process ABC invokes the READ process (a system input process) which when executed causes system data to be transferred into the READ process. The system data then becomes process data local to the READ process. This data is transferred via the output interface of the READ process to the ABC process which assigns the data to its interior data names A, B, and C. Next, the XYZ process is invoked with data names ARF and C. The data which is associated with the data names ARF and C and local to the ABC process are transferred via the XYZ process input

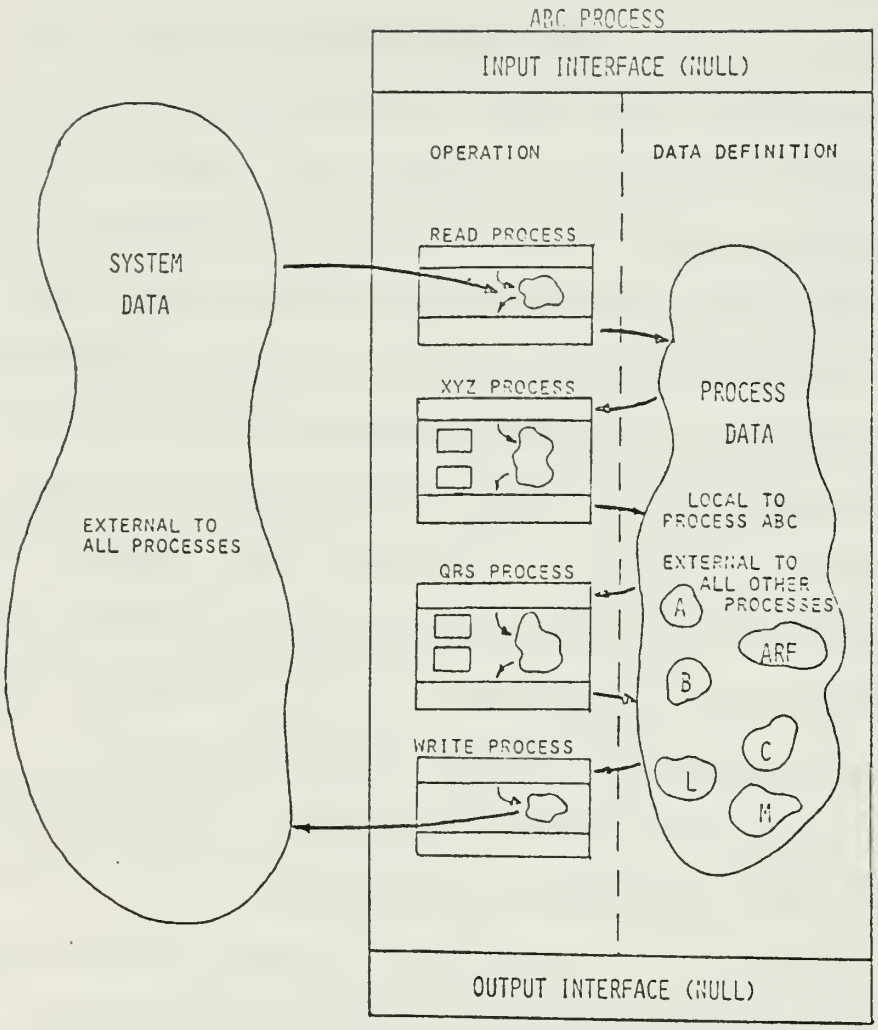


Figure 4-9

Example IV - 7: A process which corresponds to Figure 4-9:

```
ABC PROCESS
DATA
     ARF, L, M: CHARACTERS;
     A, B, C: NUMBERS;
END DATA
OPERATION
     A, B, C←READ;
     L, M←XYZ(ARF, C);
     ARF←QRS(L, M, B);
     WRITE (L, M, A, B, C, ARF);
END OPERATION
END PROCESS
```

interface to the local pool of process data within the XYZ process. The data (ARF and C) transferred to the XYZ process can now be used by the XYZ process. At the completion of the execution of the XYZ process data is transferred out of the XYZ process via the output interface into the ABC process in a manner similar to that described for the READ process. The same transfer of data and changing of local environment occurs with the QRS process and the WRITE process. Once data has become local to the WRITE process it is transferred to the external system environment and becomes systems data.

The above discussion of Example IV - 7 has very informally indicated some rules concerning the scope of data names. The scope of a name is that portion of the process over which the name can be used. At any point during the execution of a process there exists a set of active associations between names and process or data. The point at which a particular name association is active during execution of the process is determined by the scope rules.

The base language has two basic scope rules. First, the scope of data names defined in the process data definition section consists of that process. That is, a data name may be referenced only within the process in which it is defined. Second, the scope of a process name consists of all processes which have been declared (i.e., are grouped together for translation) or are in the system library.

SYNTAX DIAGRAM

PROCESS

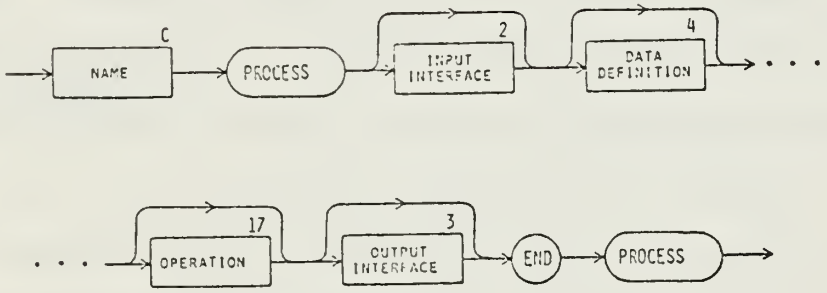


Figure 4-10

Example IV - 8: Determination of the maximum of two numbers

```
MAX PROCESS              # PROCESS DECLARATION
  INPUT                  # INPUT INTERFACE SECTION
    A, B
  END INPUT
  DATA                   # DATA DEFINITION SECTION
    A, B: NUMBERS;
  END DATA
  OPERATION              # OPERATION SECTION
    (A>-B): DEPART;      # IF TRUE GO TO OUTPUT SECTION
    A←B;
  END OPERATION
  OUTPUT                 # OUTPUT INTERFACE SECTION
    A
  END OUTPUT
END PROCESS
```

b. Process

The process is the primary component of the base language. It is a translatable unit which is executable if all required data is contained within the process or the data it requires is made available by another process. Example IV - 8 shows a process which requires two data values to be passed into it in order to execute properly. The process is a closed unit which has only one entry point and only one exit point.

The structure of the process is shown in Syntax Diagram 1 and in Example IV - 8. The name of the process is a user defined identifier which must be unique. The structure of a name is described in Section IV.C.7.c.(1). The body of the process consists of four optional sections and is enclosed by the reserved words PROCESS and END PROCESS. The sections of the process, in the order in which they must be specified are: (1) the input interface section which is described in Section IV.C.3.b, (2) the data definition section which is described in Section IV.C.4, (3) the operation section which is described in Section IV.C.5, and (4) the output interface section which is described in Section IV.C.3.c. Each of these sections is optional and need not be specified if not required by a particular process. An example of a process which does not require an input interface section would be a process that uses only internal data. In this case the input interface section of the process need not be specified.

3. Process Interfaces

a. Basic Concepts

The process interfaces act as channels for the transmission of data between processes. The concept of interfaces was developed to insure that unwanted side effects cannot occur from internal process data manipulation. The interface defines the data name, the number of data items and the order of the data items. The interfaces can be thought of as electronic multipin connectors as depicted in Figure 4-11

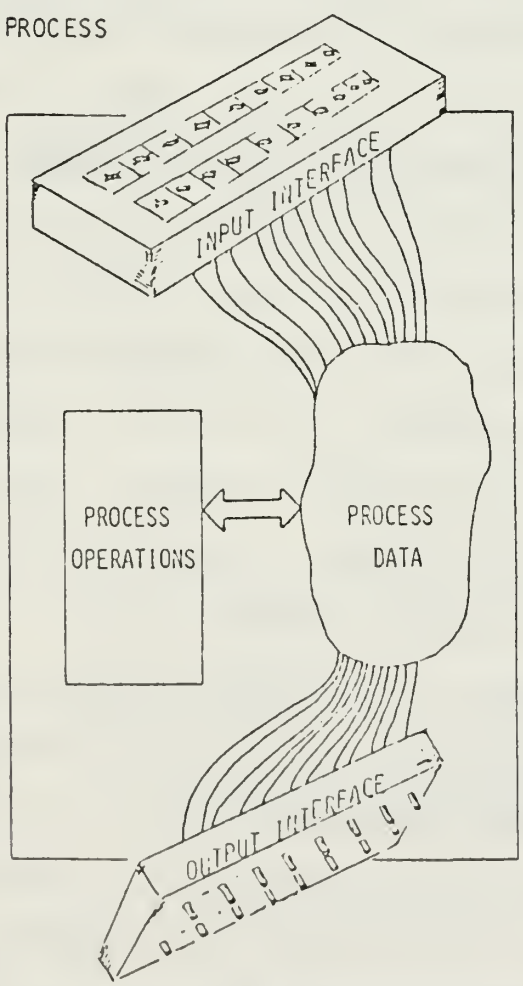


Figure 4-11

The input interface provides a channel for data leaving the process to an external source. All external data must enter the process through the input interface. The output interface provides the only exit for internal process data. A difference between the base language and many existing computer languages is that in the base language multiple data items may be passed out of a process.

In order to maintain the concepts of a closed process and ensure no side effect contamination of data, the value of the actual parameter must be protected from modification during the execution of the invoked process. The implementation of the input and output interface mechanisms should provide this protection.

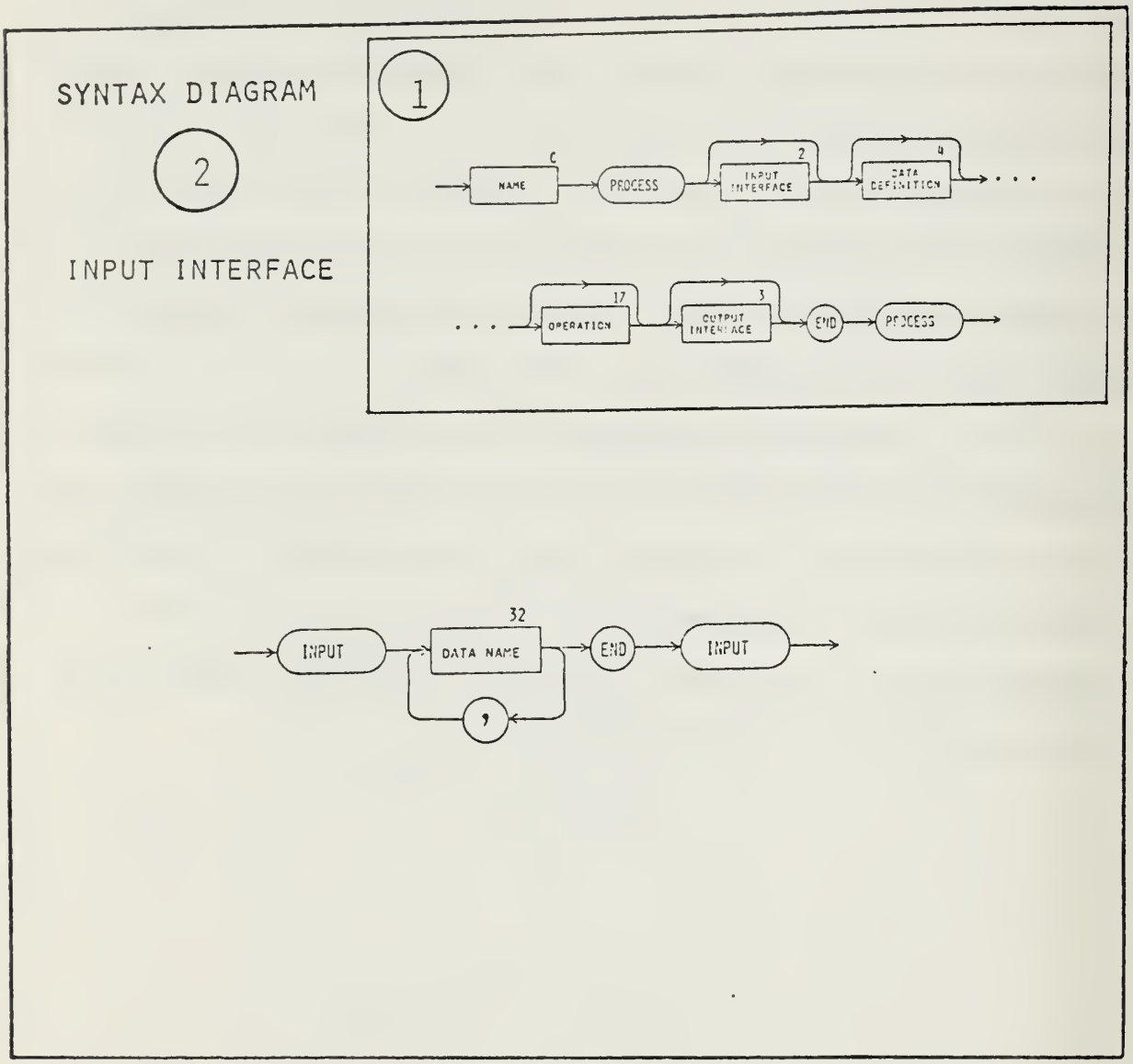


Figure 4-12

Example IV - 9: Sample Input Interface Code

```
INPUT
  A,B
END INPUT
```

b. Input Interface

The syntactic structure of the input interface is depicted in Syntax Diagram 2, and in Example IV - 9. The input interface section consists of a list of data names

separated by commas. The list is surrounded by the reserved words INPUT and END INPUT which mark the beginning and end of the section.

The input interface section is the first section in a process. If the process does not require external data then the input interface section may be omitted. An omitted input interface section is referred to as a null input interface.

The data names listed in the interface are associated with the values of the data names in the process invocation parameter list (Section IV.C.5.d.(4)(a)) when the process is invoked. In Example IV - 9 the value of each input parameter (A and B) is given an initial value equal to the associated outer process data name value. The initial value of A is the value of the first actual parameter passed during invocation and the initial value of B is the value of the second actual parameter. In the invoked process, operations are performed on the values of A and B, thus protecting the values of the actual parameters.

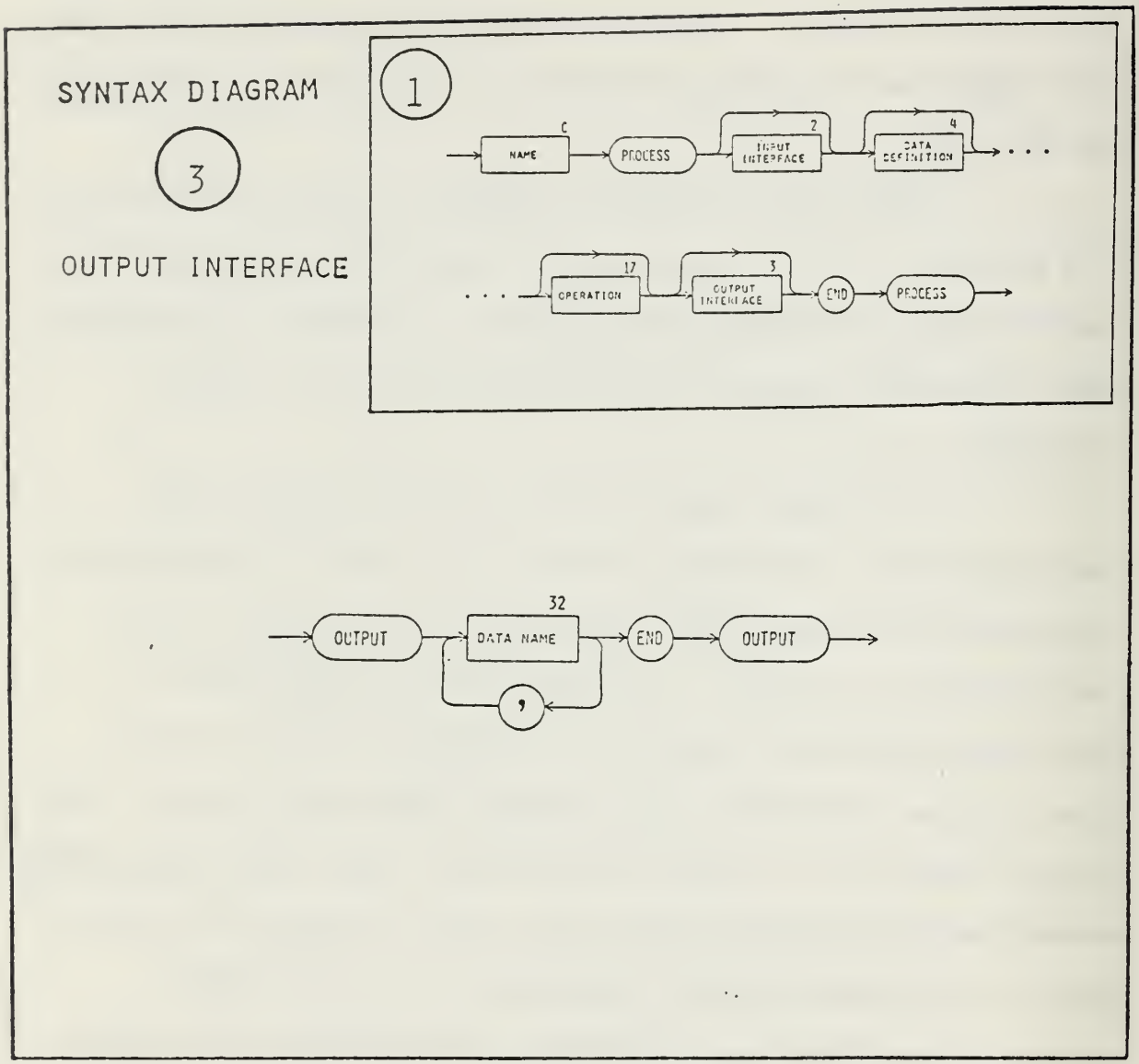


Figure 4-13

Example IV - 10: Sample output interface code

```
OUTPUT
  A
END OUTPUT
```

c. Output Interface

The syntactic structure of the output interface is depicted in Syntax Diagram 3, and in Example IV - 10. Its structure and contents are very similar to those of the

input interface. The output interface section consists of a list of data names separated by commas. The list is surrounded by the reserved words OUTPUT and END OUTPUT which mark the beginning and end of the section. Example VI - 10 is the output interface section from Example IV - 8 and shows a list of one data name. If no data is passed out of a process via the interface, the output interface section may be omitted from the process. In this case the output interface is referred to as a null output interface.

The output interface section is the last section in a process. The data names listed in this section represent the values which are passed out of a process for use by other processes. These values may be assigned to data names in the calling process through the naming operation (Section IV.C.5.c.), used directly by the calling process through the answer operation element (Section IV.C.5.d.(4)(d)), or ignored by the calling process altogether by not assigning or using them in conjunction with process invocation (Section IV.C.5.d.(4)(a)).

4. Data Declaration

a. Basic Concepts

All data used within a process must be identified and the structure or property of each data item must be described. The data definition section of a process provides the vehicle for specifying this information.

The identification of data is accomplished by providing a name which can be associated with the data item. The property of a data item describes the structure of the data item. Built-in structures such as ARRAY, LIST, and SET, and the basic data elements BIT and BYTE are considered primitive structures and are provided by the base language. Through the use of these built-in structures and elements, the user may define the data properties which are necessary for this particular application or implementation of the base language. For example, an integer may be defined as a single dimension array of 16 bits in the following manner; INTEGER: ARRAY [16] OF BITS;. Now, INTEGER can be used as a data element to define the properties of other data items such as ONE: INTEGER; or NUMBERLIST: ARRAY [10] OF INTEGER;. In this manner the user can define application oriented data properties in terms of properties which have been defined previously.

In addition to defining the name and property of a data item the user can optionally define data value control mechanisms which are called amplifying information. These mechanisms provide the following capabilities:

(1) initial or constant value assignment, (2) bounds on valid data values, (3) run time error checking of data values, and (4) user defined exception handling procedures. Exception procedures are executed when the value of the data item is not within specified tolerances. The use of these mechanisms is described in the discussion of amplifying information in Section IV.C.4.e.

A user defined data structure may be saved in the system library through the use of the LEARN directive. This allows application or implementation oriented structures to be defined once and saved for future use. These learned structures may then be referred by name when needed. Any amplifying information associated with a saved structure can be overridden when the structure is used by specifying new amplifying information. For example, suppose the following data description of a basic data unit was defined and stored in the system library:

```
INTEGER: ARRAY [16] OF BITS, LIMIT (0..65535);.
```

This description is dependent on the type of machine the base language is implemented on, and represents only one of many descriptions which could be used to define a group of natural numbers. The example is presented to show that basic data definitions may be created and saved for future reference. This description could then subsequently be used to describe a data item in a process requiring different value limits by referencing the saved

name in this manner: NEWNUM: INTEGER, LIMIT (0..100);. The new limits override the originally defined limits for INTEGER and apply only to the data name NEWNUM.

As discussed previously, the scope of data names is limited to the process in which they are defined. There is no globally accessible data in the base language. All data must be transmitted through the process interface. When a data item is passed, its property in both the calling and called process must be the same. As a user convenience to avoid specifying the same property many times in various processes, the property of a data item can be "imported" into a process when it is called. This is done by specifying that the property is IMPORTED in the data description, for example, ANOTHERNUM: IMPORTED NEWNUM;. This feature is described in detail in Section IV.C.4.d.(3).

THIS PAGE INTENTIONALLY LEFT BLANK

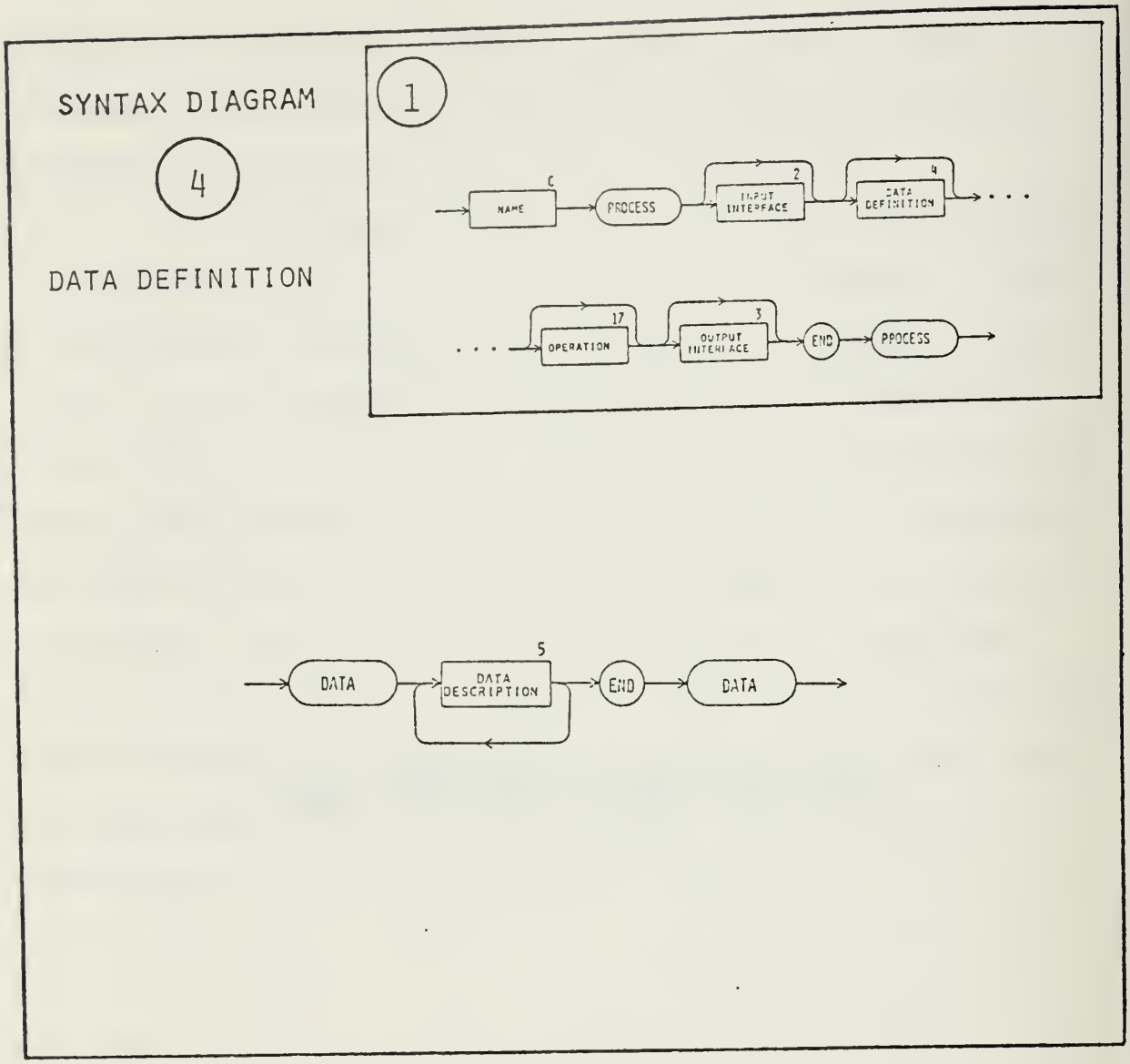


Figure 4-14

Example IV - 11: Simple data description section

```
DATA
  A, B: INTEGER, INITIAL VALUE (0);
  C: CHARACTER, LIMIT (A..Z), CERROR;
  NUMLIST: ARRAY [5] OF INTEGER, INITIAL VALUE (0,1,2,3,4);
END DATA
```

b. Data Definition

The data definition section consists of a sequence of data descriptions surrounded by the reserved words DATA and

END DATA which mark the beginning and end of the section. The structure is depicted in Syntax Diagram 4 and Example IV - 11. A complete explanation of Example IV - 11 is contained in the following sections.

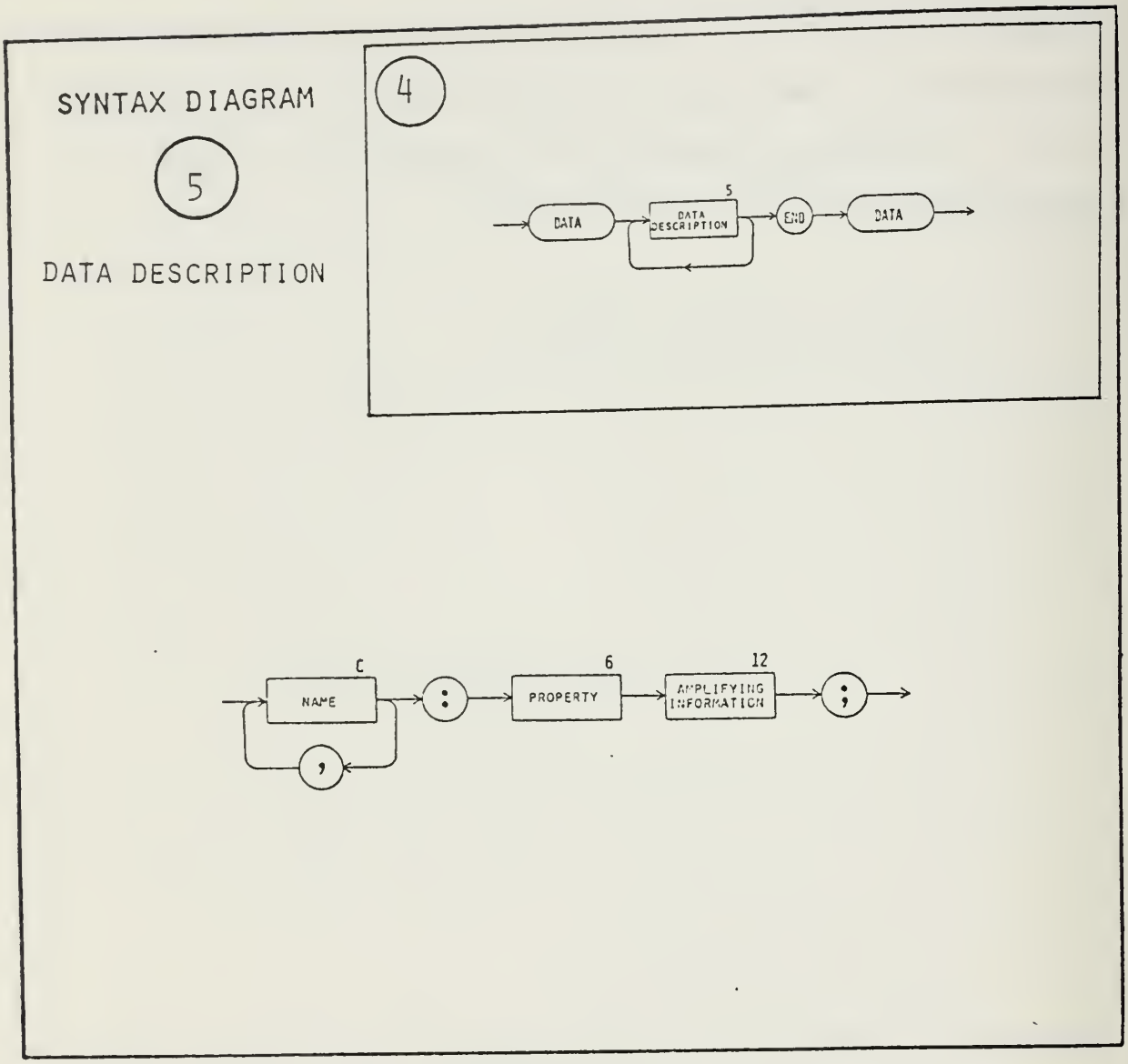


Figure 4-15

Example IV - 12: Sample data descriptions

```
    ANYNAME, ANOTHERNAME: INTEGER, INITIAL VALUE (10),
LIMIT (0..100), ERRPROC;
    !LEARN! INTEGER: ARRAY [16] OF BITS, LIMIT (0..65535),
OVERFLOW;
```

c. Data Description

The syntactic structure of a data description is shown in Syntax Diagram 5. The identification portion of

the description, to the left of the colon symbol, consists of a name or a name list which is a sequence of names separated by commas. A single name preceded by the LEARN directive (Section IV.C.6.a) causes the data description to be saved in the system library for subsequent use. Example IV - 12 shows both of these constructs.

The information to the right of the colon symbol describes the property and any amplifying information associated with the name or names. The property describes the structure of the associated name or names and is discussed in detail in Section IV.C.4.d.. The property in the first data description of Example IV - 12 is ARRAY [16] OF BITS and in the second description it is INTEGER. A property must be specified for all data names. The amplifying information associated with the name list is optional and provides a capability to assign values to data names such as INITIAL VALUE (10) in the second data description of Example IV - 12, specify limit on the values and to specify exception handling routines to be called in case of an error such as LIMIT (0..65535), OVERFLOW in Example IV - 12. Amplifying information is described in detail in Section IV.C.4.e..

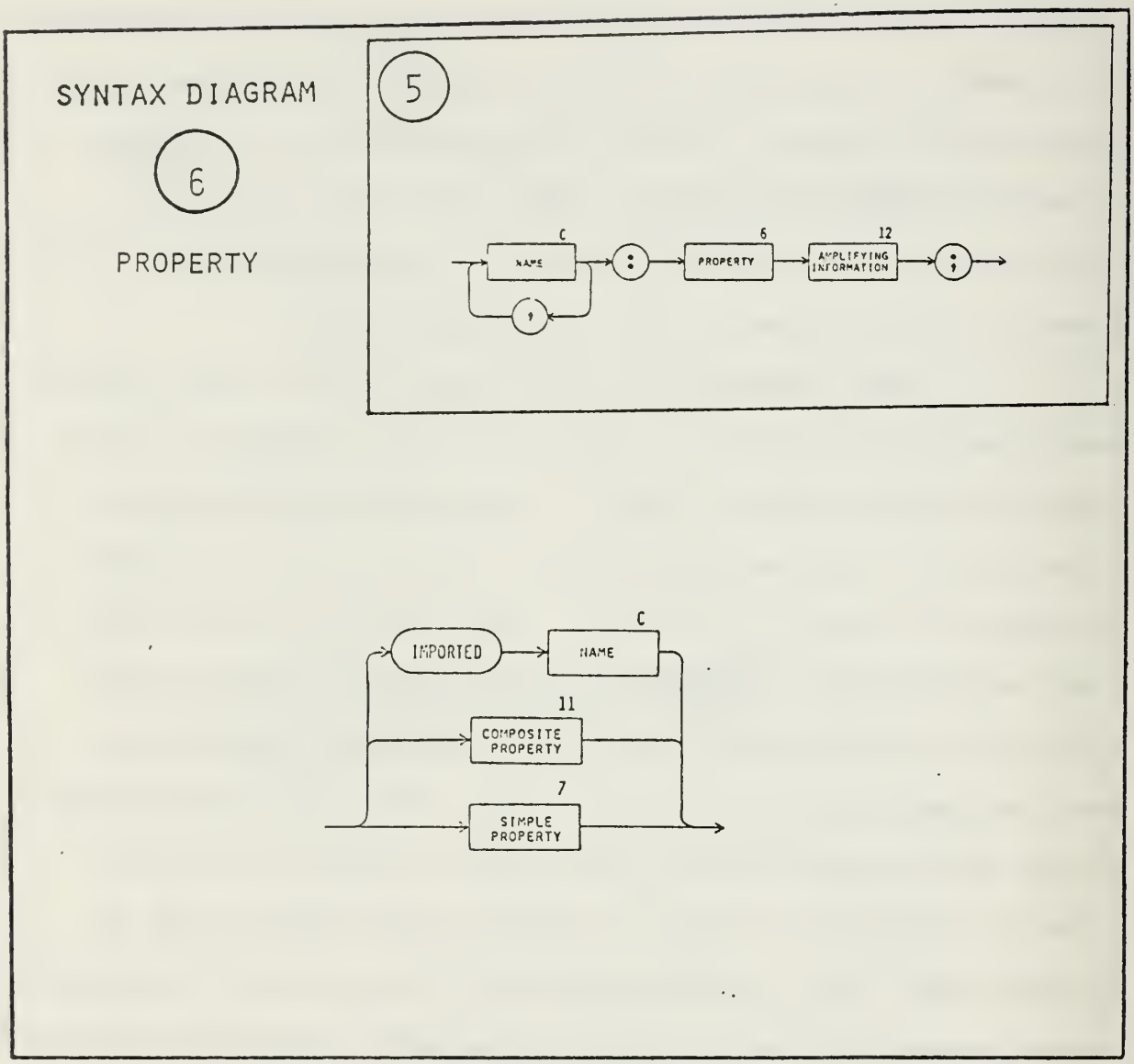


Figure 4-16

d. Properties of Data Names

There are three categories of properties; simple properties, composite properties and imported properties. Simple properties are used to describe homogeneous data structures such as arrays, lists or sets. All elements of a simple property must be of identical structure. Composite properties are used to describe non-homogeneous structures.

The individual elements of a composite property may be arbitrarily different from one another. An imported property is used to define a property which is associated with a data name which is being passed into a process via the input interface. Each of these categories is discussed in detail in the following sections.

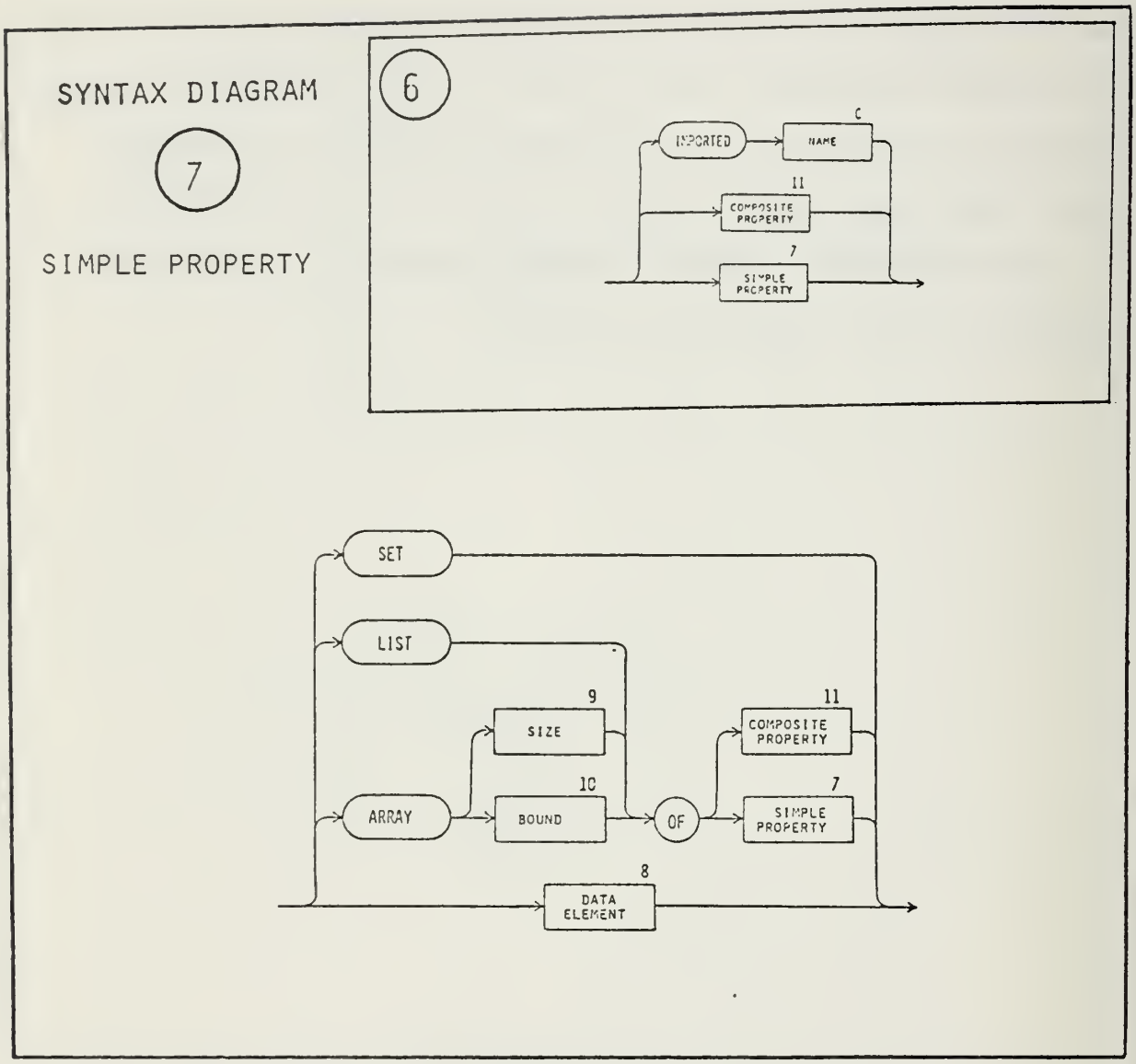


Figure 4-17

(1) Simple Property. The simple property provides the primitive structures which can be used to build abstract application or specific implementation oriented data structures. The syntactic structure of a simple property is shown in Syntax Diagram 7. As can be seen from the diagram the most basic property is the data element which is described in Section IV.C.4.d.(1)(a).

The ARRAY and LIST properties are described in Sections IV.C.4.d.(1)(b) and (c) respectively. The recursive definition allows very complex structures to be developed, such as, arrays of arrays or list of lists of arrays, etc. The SET property is defined in Section IV.C.4.d.(1)(d). All structures defined by the simple property are homogeneous in nature and must, therefore, have identical form. The imported property may not be used in the recursive definition of arrays or lists.

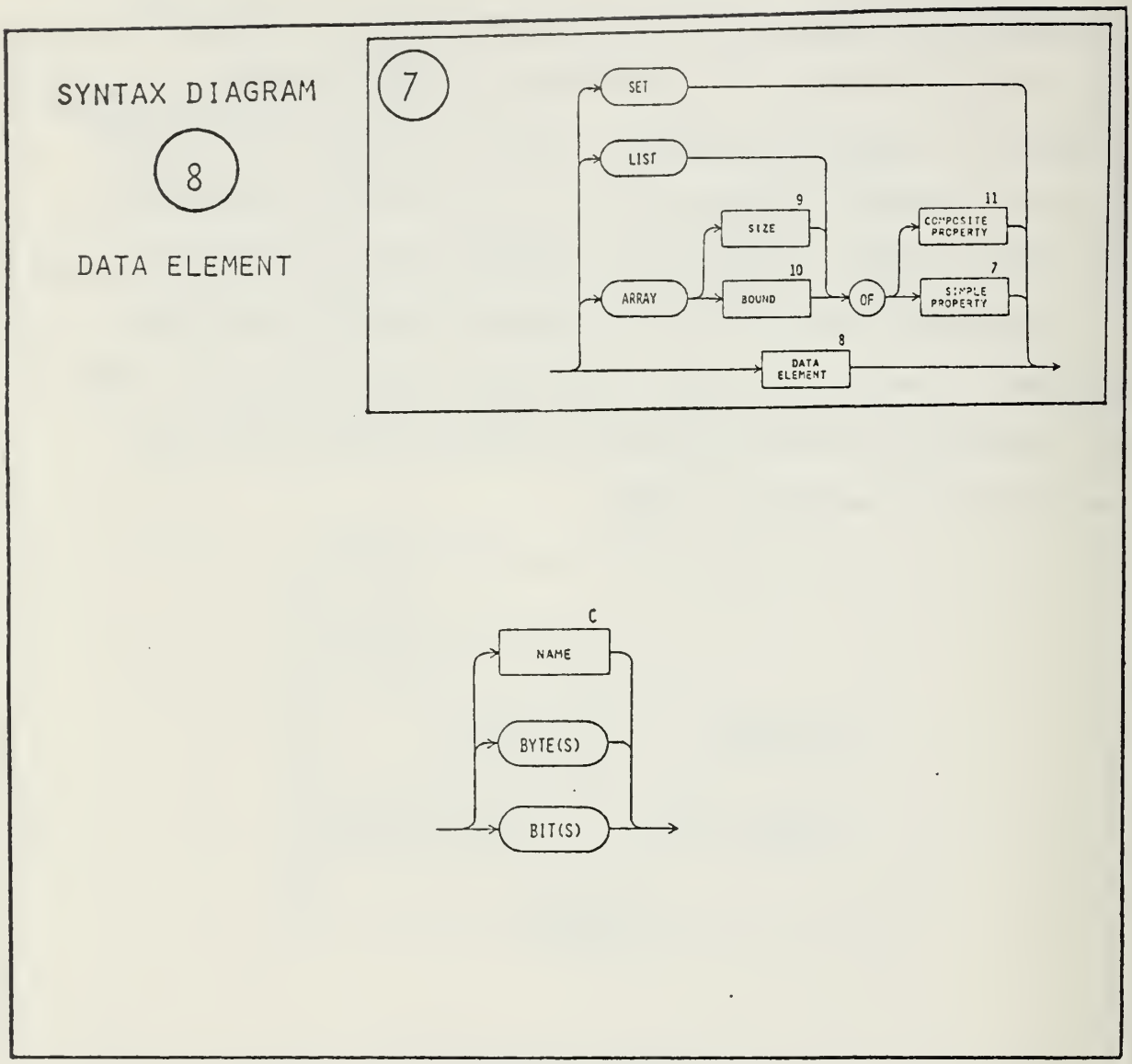


Figure 4-18

Example IV - 13: Sample code of data description using a data element property

```
TRUE: BIT, CONSTANT VALUE (1);
```

(a) Data Element. The data element is the most basic data property. It provides extensibility of data structures through the use of the two basic elements

BIT or BYTE and user defined elements. Large complex structures can be built by incrementally building up from the basic elements and user created elements. The LEARN feature then allows these structures to be reused without rebuilding or respecifying.

The reserved words BIT or BITS and BYTE or BYTES represent one computer memory bit and one byte composed of eight bits, respectively. The name in the definition is a user defined name which has been associated with a property and possibly with amplifying information through a data description. BIT and BYTE are the only pre-defined data elements in the base language. All other data elements such as integer, real, boolean, character, etc., must be constructed from the basic elements or user defined elements. This provides machine independence of the base language and the ability of the user or implementer to structure application oriented data elements in accordance with the target machine specifications.

Example IV - 13 shows a data item named TRUE which has the property of BIT and a constant value of one. Another manner of defining the data name TRUE would be to first define a name BOOLEAN with a property of one bit such as BOOLEAN: BIT;. This description could be saved in the system library and referred to in defining other data names with the bit property such as TRUE: BOOLEAN, CONSTANT VALUE(1); or FALSE: BOOLEAN, CONSTANT VALUE (0);.

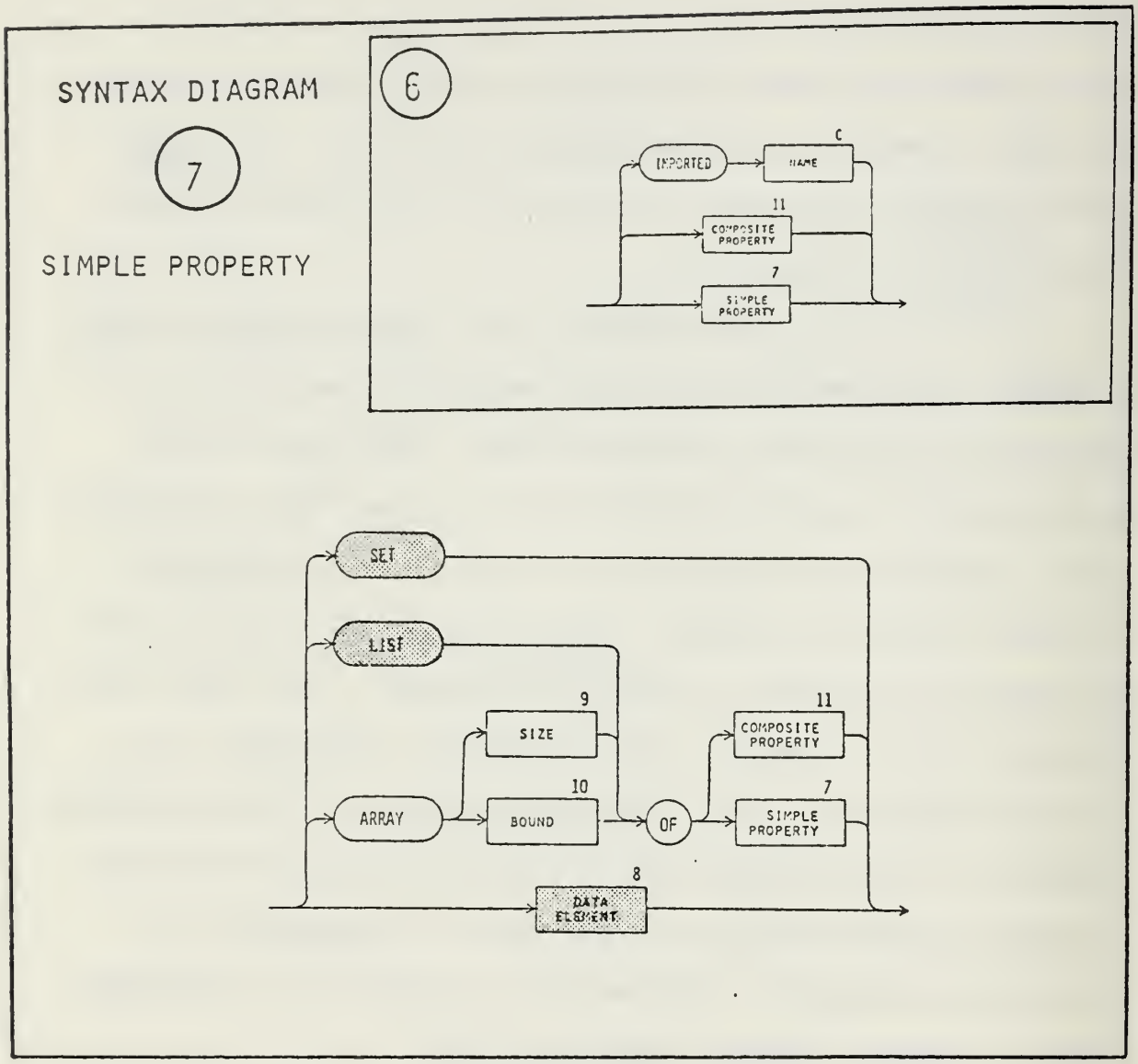


Figure 4-19

Example IV - 14: Sample data description using array property and size dimension

```
ALPHA: ARRAY [3 BY 4] OF INTEGER;
```

Example IV - 15: Sample data description using array property and bounds dimension

```
BRAVO: ARRAY [1..12] OF INTEGER;
```

(b) Array Description. The array property is similar to array types in other languages and is an element of the syntactic category "simple property". It is a built-in structure provided by the base language. The array describes a group of data items which are naturally ordered in some fashion. It consists of a fixed number of units, the number of which is defined by the size or bounds of the array. All units of the array have the same homogeneous base property. In Examples IV - 14 and IV - 15 the base property is INTEGER.

The dimension of the array, which indirectly determines the number of units in the array, is defined by its size (Section IV.C.4.d.(1)(b)[1]) or bounds (Section IV.C.4.d.(1)(b)[2]) of the array. A one demensional array is a vector and is shown in Example IV - 15. Example IV - 14 provides a sample of a two dimensional array. The number of dimensions which an array may have is not limited by the base language definition but may be restricted by the implementation of the language.

Array units may consist of any property structure except the IMPORTED property. For example: array of data element, array of array of data element, array of list of data element and so on. Each individual unit of an array may be explicitly denoted and directly accessed by using the array name followed by an index value. Array accessing is described in Section IV.C.5.d.(4)(b).

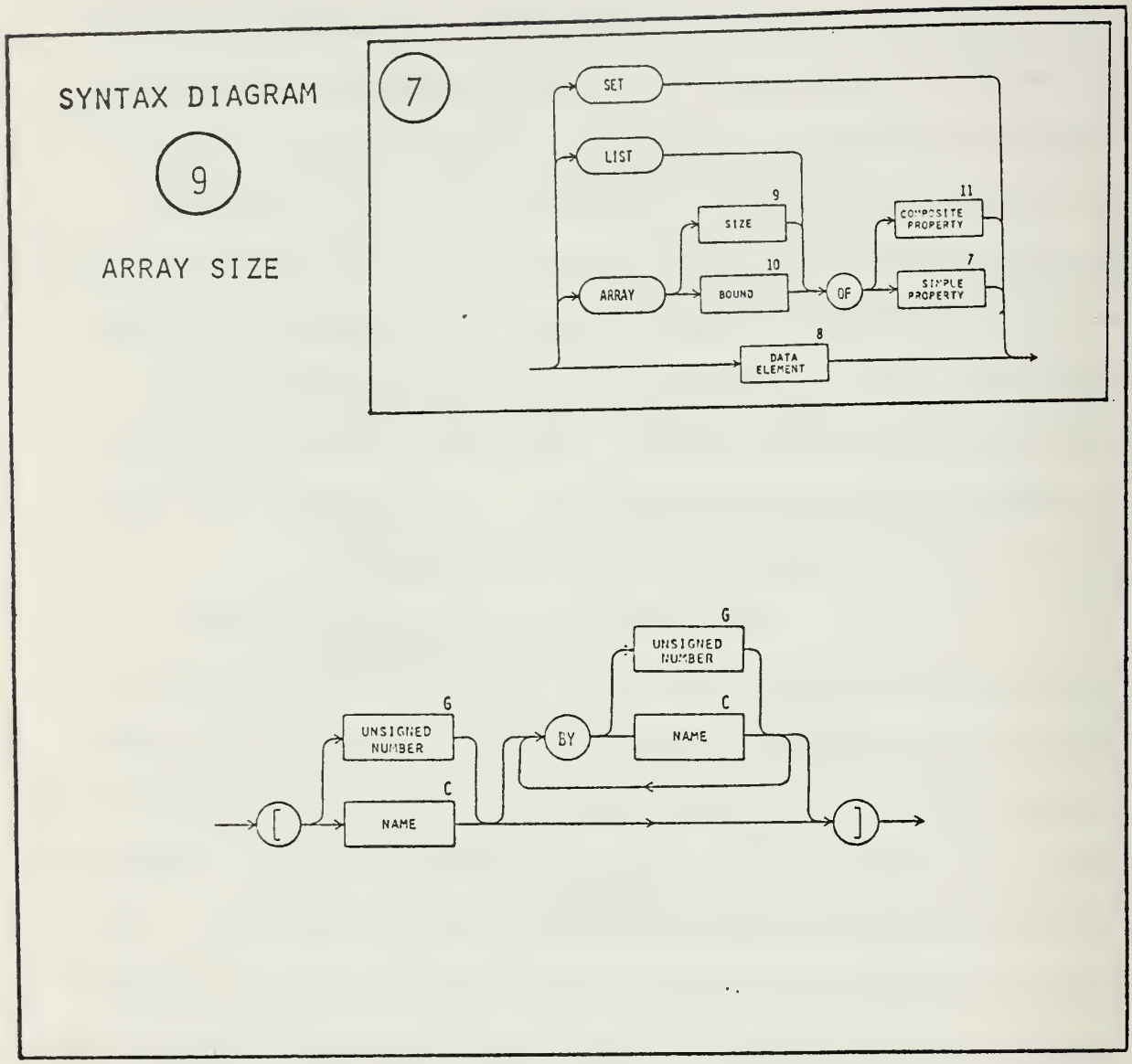


Figure 4-20

[1] Array Size. The array size specifies the dimension and number of units in an array. The syntactic structure is shown in Syntax Diagram 9. The size of each individual dimension is represented by an unsigned number or a name which indicates the number of units in that dimension. The total size of the array may be represented by a sequence of unsigned numbers or names separated by the

reserved word BY. The size is enclosed in square brackets. The number of dimension in an array is determined by the number of elements in the sequence which defines the size. Example IV - 14 depicts a two dimensional array. The number of units in an array is determined by multiplying all elements in the size sequence. The array described in Example IV - 14 contains 12 units. An array size which is defined using real or fixed point numbers such as [2.135 BY 6.141] is syntactically legal and should be interpreted as a two dimensional array of 12 units with an implied size of [2 BY 6]. All fractional parts of an unsigned number are truncated when the size is evaluated. When using the array size description to describe an array, the index numbers used to access the array elements are implementation dependent.

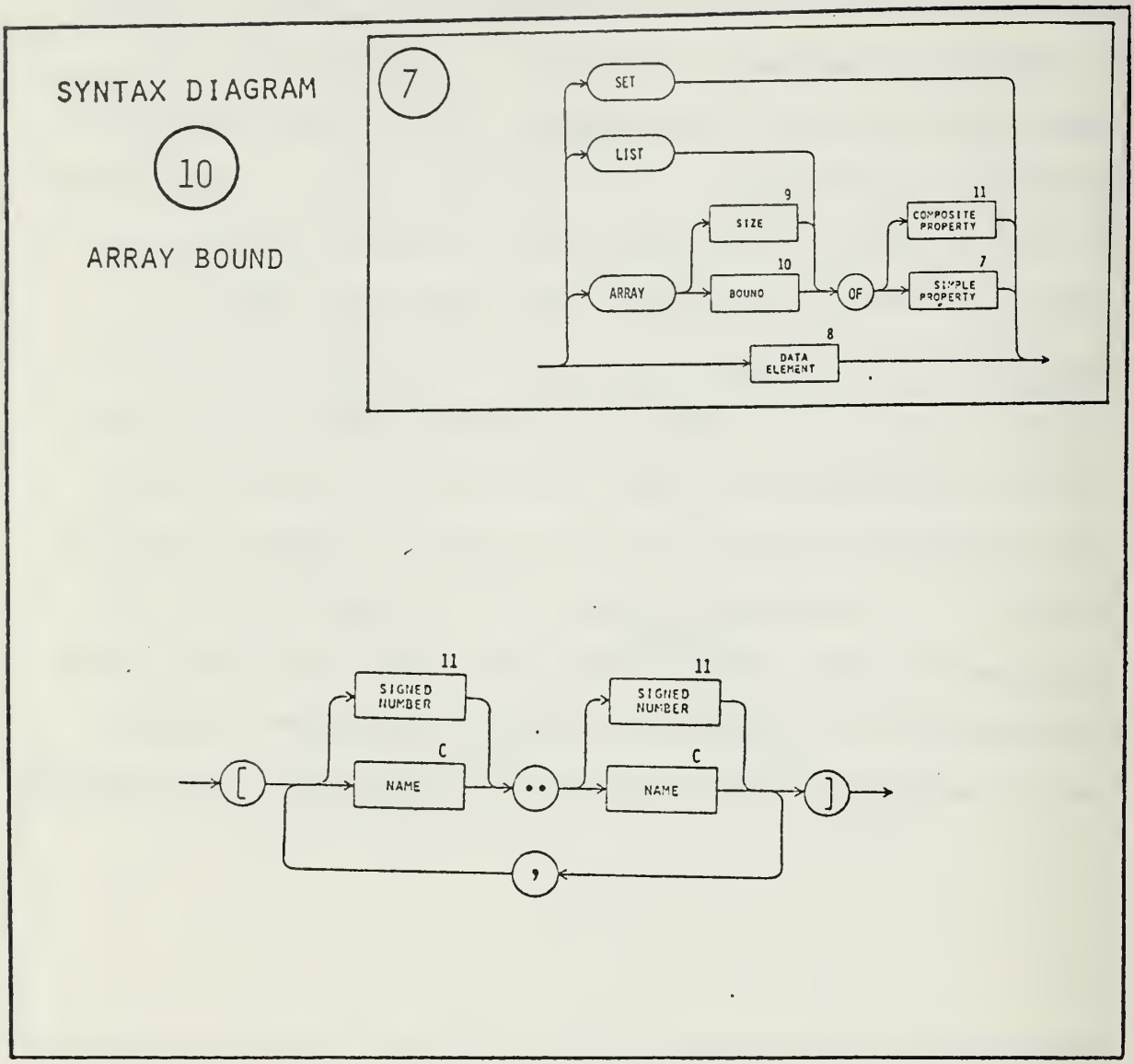


Figure 4-21

Example IV - 16: Sample of array bounds

```
[N..M]
[-5..15]
[A..B, B..X, 5..10]
```

[2] Array Bounds. The array bound represents the dimensions of the array by specifying bounds on the index numbers used to access individual units of each dimension in the array. The structure of the array

bounds is shown in Syntax Diagram 10. It consists of a sequence of bound pairs separated by commas and enclosed between square brackets. A bound pair consists of a starting index value and an ending index value represented by a name or a signed number separated by two contiguous periods. The starting index value must be less than the ending index value. Example IV - 16 presents some sample array bounds.

The number of dimensions in an array is indicated by the number of array bound pairs specified. The size of each dimension is determined by subtracting the beginning index value from the ending index value and adding one to the absolute value of the result. As in the array size (Syntax Diagram 9), when bounds are specified using real or fixed point numbers, the fractional portion of the number is truncated when the bound is evaluated. The array bound presented in Example IV - 16 depicts a single dimension array with |M - N| + 1 units. The second example represents a one dimension array with 21 units. The third example represents a three dimensional array with (|B - A| + 1)*(|X - B| + 1)*6 units.

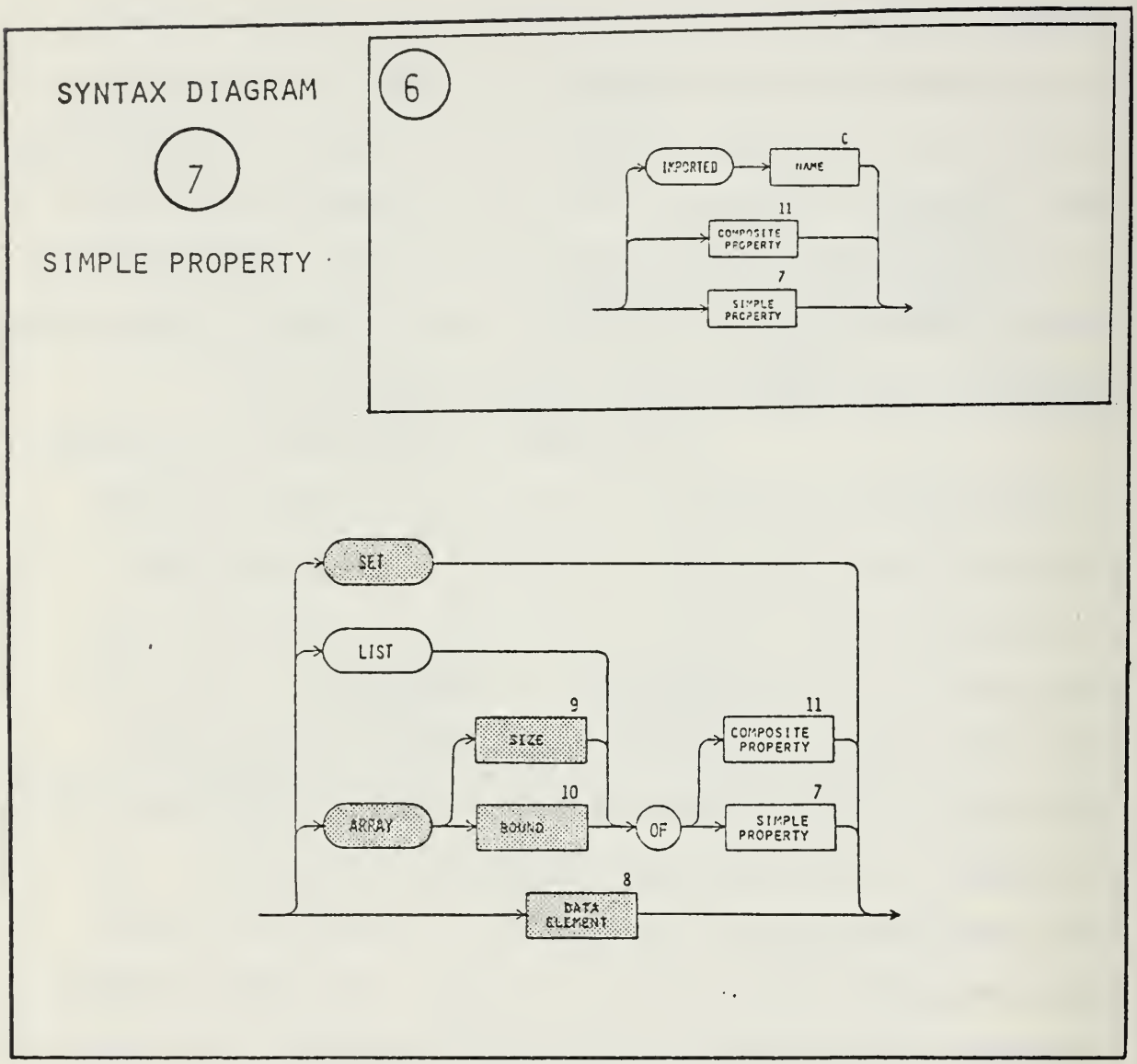


Figure 4-22

Example IV - 17: Sample list data description

```
NUMLIST: LIST OF REAL;
KEYWORDS: LIST OF ARRAY [5] OF CHARACTER;
```

(c) List Description. The list structure is a dynamic structure which consists of a logically ordered set of units. These units are chain linked together by a series of pointers. The unit, sometimes referred to as a

node, contains one or more pointers, depending on implementation, and a value field. The structure of a list is shown in Syntax Diagram 7, Figure 4-22. The pointer which points to the head of the list is accessed through the use of the list name. All pointers are transparent to the user. The structure of the value field is defined by the property following the reserved words LIST OF. The value field may be any property from a data element to a complex structure but it may not be an IMPORTED property. The nodes of a list are accessed by special list operators which are described in Section IV.D..

Example IV - 17, first presents a simple list data description. The list name is NUMLIST. A reference to this name provides access to the head of the list. The value field of each node consists of a real number (this assumes that REAL has been previously defined). The second description specifies a list of one dimensional arrays, each containing five units. Each unit of the array has the previously defined property CHARACTER.

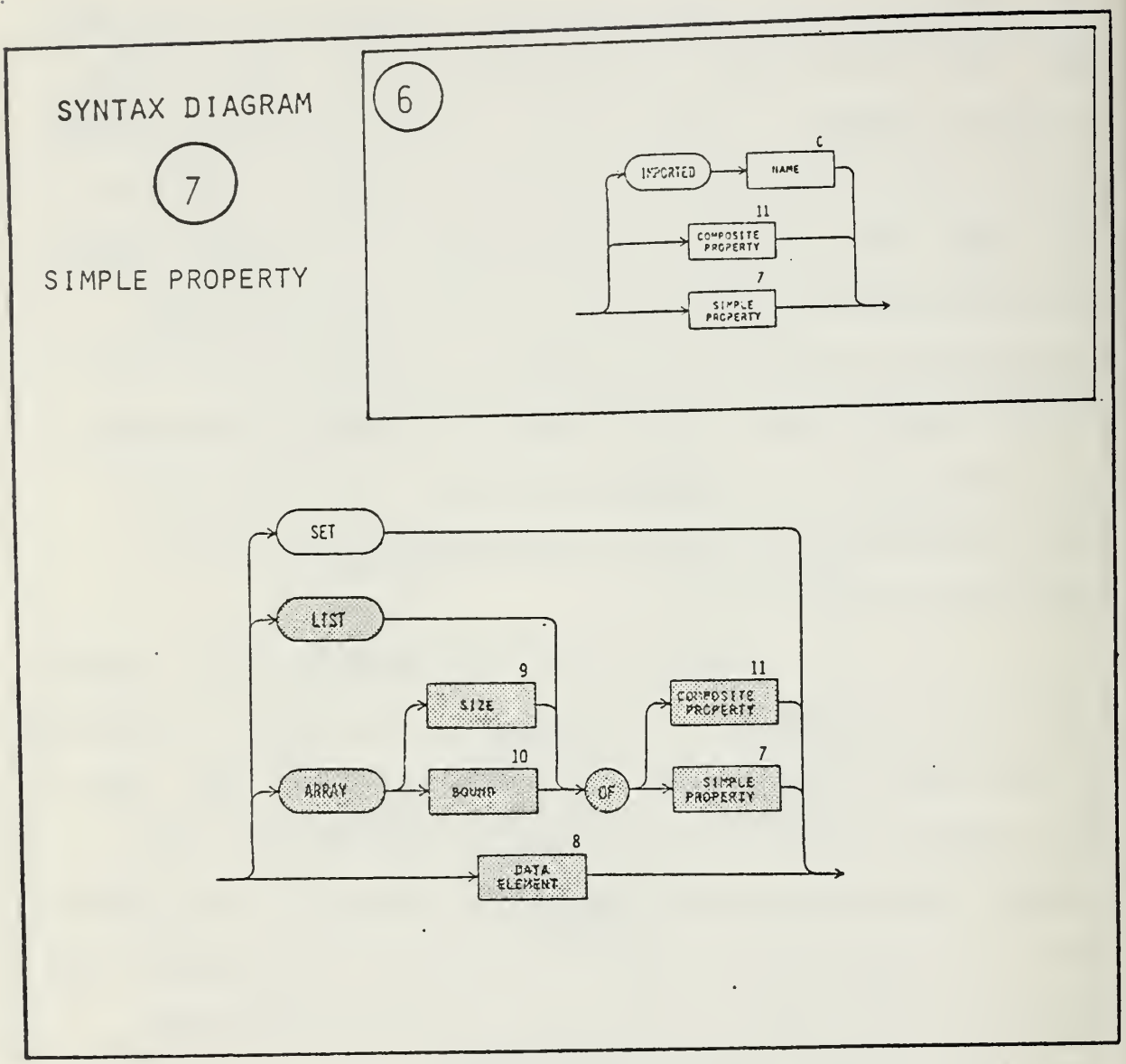


Figure 4-23

Example IV - 18: Sample of set description

```
CHARACTERSET: SET, INITIAL VALUE ('A', 'B', 'C');
FLAGCOLORS: SET, CONSTANT VALUE (RED, WHITE, BLUE);
```

(d) Set Description. The set property defines a grouping of objects which, in a process, requires no further definition other than simple enumeration of permissible set numbers. The reserve word SET indicates to

the translator that the enumerated members present in an amplifying CONSTANT or INITIAL VALUE clause comprise a closed or open set respectively. Set operations such as addition, deletion, membership, intersection and union are examples of allowable operations on set data and are described in Section IV.D. In Example IV - 18 CHARACTERSET is an open set consisting of characters A, B, C respectively. If allowed by implementation operations, the members of such a set could conceivably be added to or deleted from during the execution of a process. FLAGSET is, on the other hand, a closed set which is defined as constant valued RED, WHITE, and BLUE. No additions or deletions could be made to such a set.

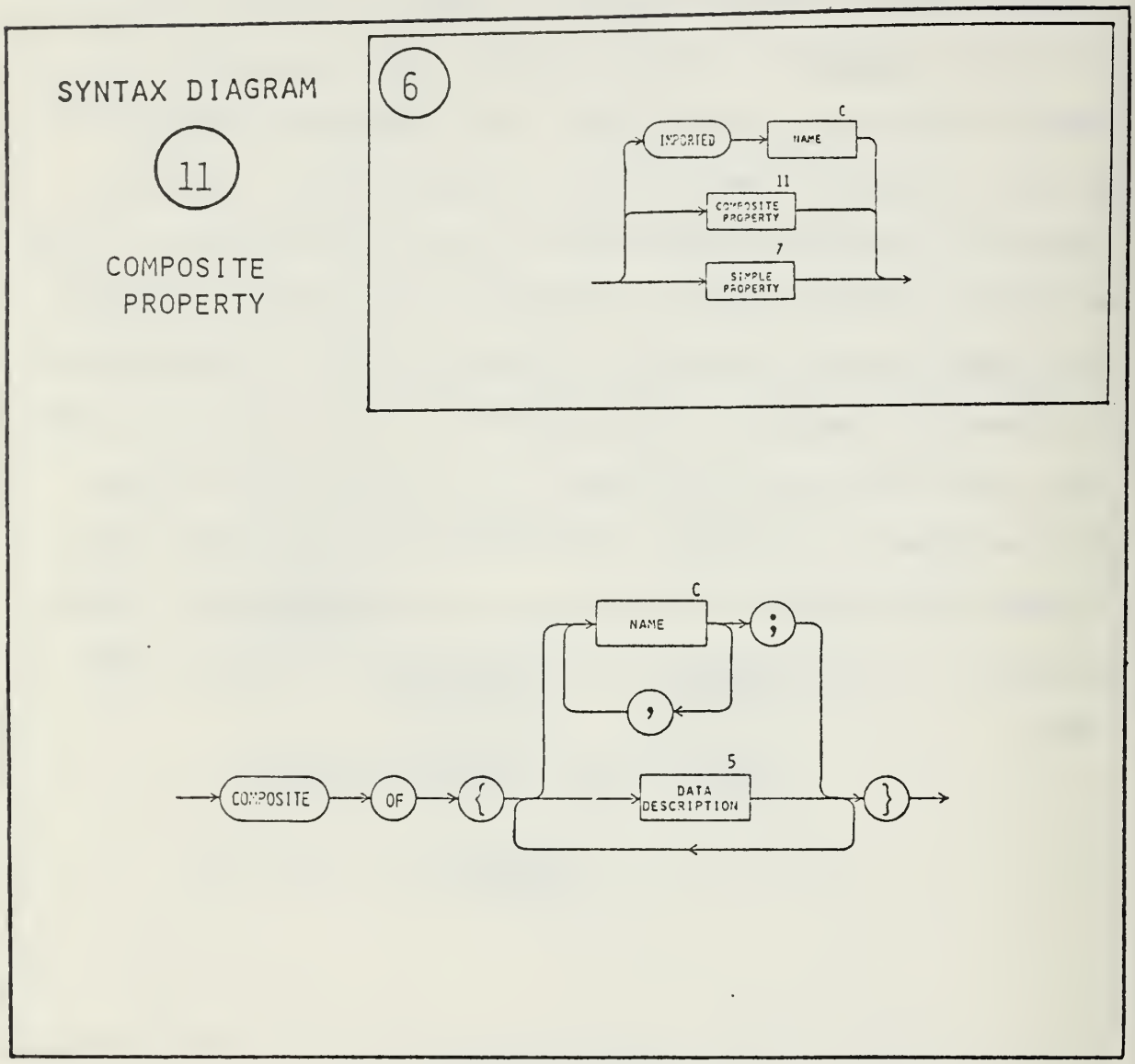


Figure 4-24

Example IV - 19: Sample of a data declaration using a composite property.

```
FORM88: COMPOSITE OF {
                      NAME: ARRAY [22] OF CHARACTER;
                      SSN: ARRAY [9] OF INTEGER;
                      OCCUPATION: ARRAY [20] OF CHARACTER;
                      CITY, STATE;
                     }
```

(2) Composite Property. The composite property is a non-homogeneous group of data structures. It is analogous to the record type of PASCAL. The syntactic structure of the composite property is shown in Syntax Diagram 11 and in Example IV - 19. The composite property consists of the reserved words COMPOSITE OF followed by a description of the composite structures' elements or fields enclosed in curly braces ({ }). The field description of a composite structure consists of either a complete data description (Section IV.C.4.c.) or the name of a data item which has previously been learned or described elsewhere in the process by a separate data description. The previously described data name may be in the same data definition section or in the system library. The fields of the composite structure may be referenced by name as described in Section IV.C.5.d.(4)(b).

Example IV-19 shows a composite structure description named FORM88 with elements NAME, SSN, OCCUPATION, CITY and STATE. The fields NAME, SSN, and OCCUPATION are complete data descriptions. The fields CITY and STATE have been previously defined and a redefinition is not required. This feature avoids unnecessary redefinitions which could lead to transcription errors.

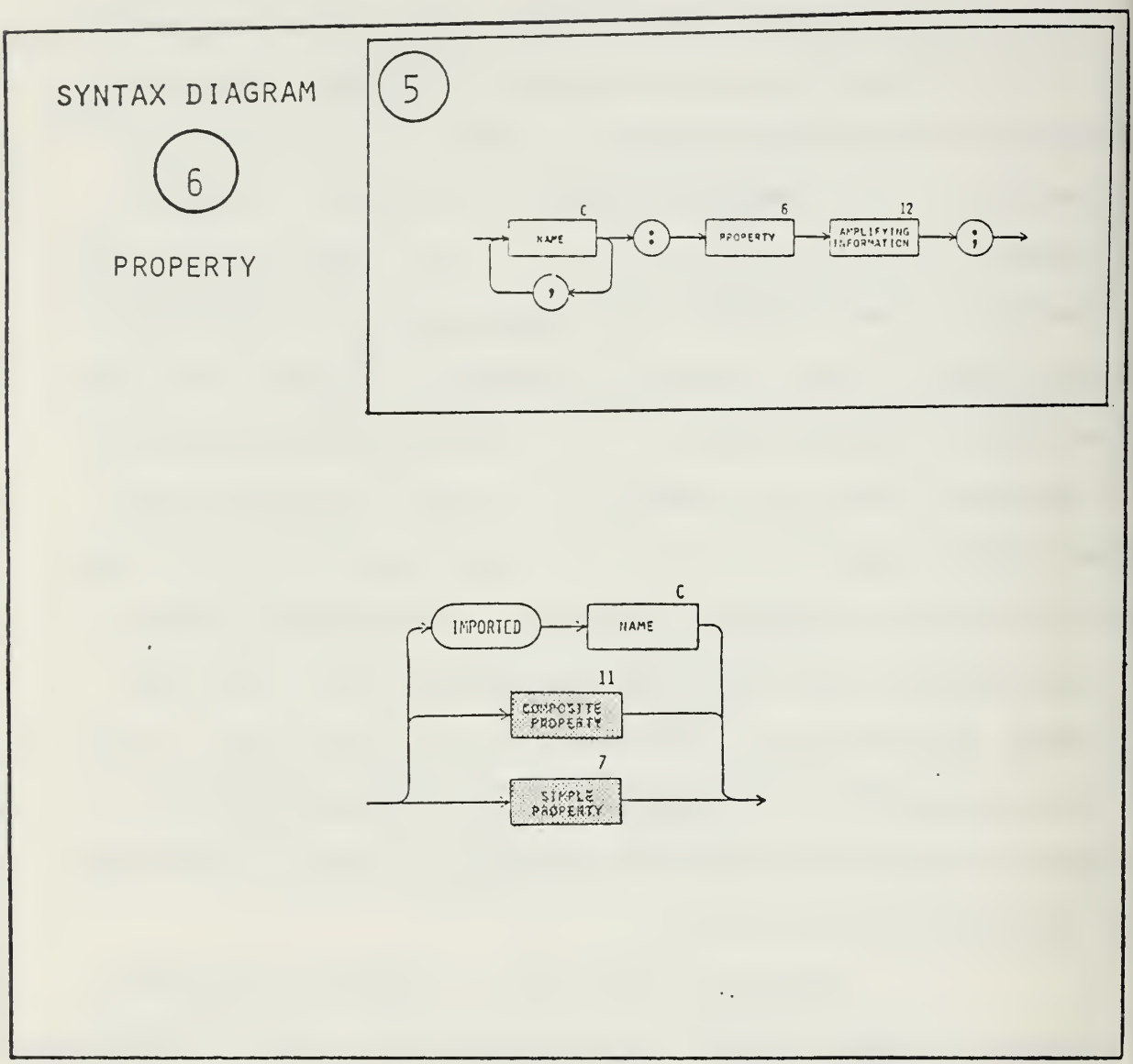


Figure 4-25

Example IV - 20: Sample of data description using an imported property

```
FORM88COPY: IMPORTED   FORM88;
```

(3) Imported Property. An IMPORTED property is a property which has been previously described in an encompassing process or outer instance. Importing a property into an interior process in effect passes the nature

of an outer data item into the data declaration of the interior process. The purpose of the IMPORTED property is to avoid unnecessary redescription which could lead to transcription errors of general system data that is to be passed from process to process. During translation, when the reserve word IMPORTED is encountered, the translator searches outwardly all encompassing processes to find a data name which follows the reserve word. When the first outward occurrence of this name is found, its property is used as the property of the data name. If amplifying information is specified with the imported property, it overrides any amplifying information present in the outward declaration. To preserve the independence of processes which are saved in the system library, a process which contains an IMPORTED property in its data definition section may not be "learned" using the LEARN directive (Section IV.C.6.a.).

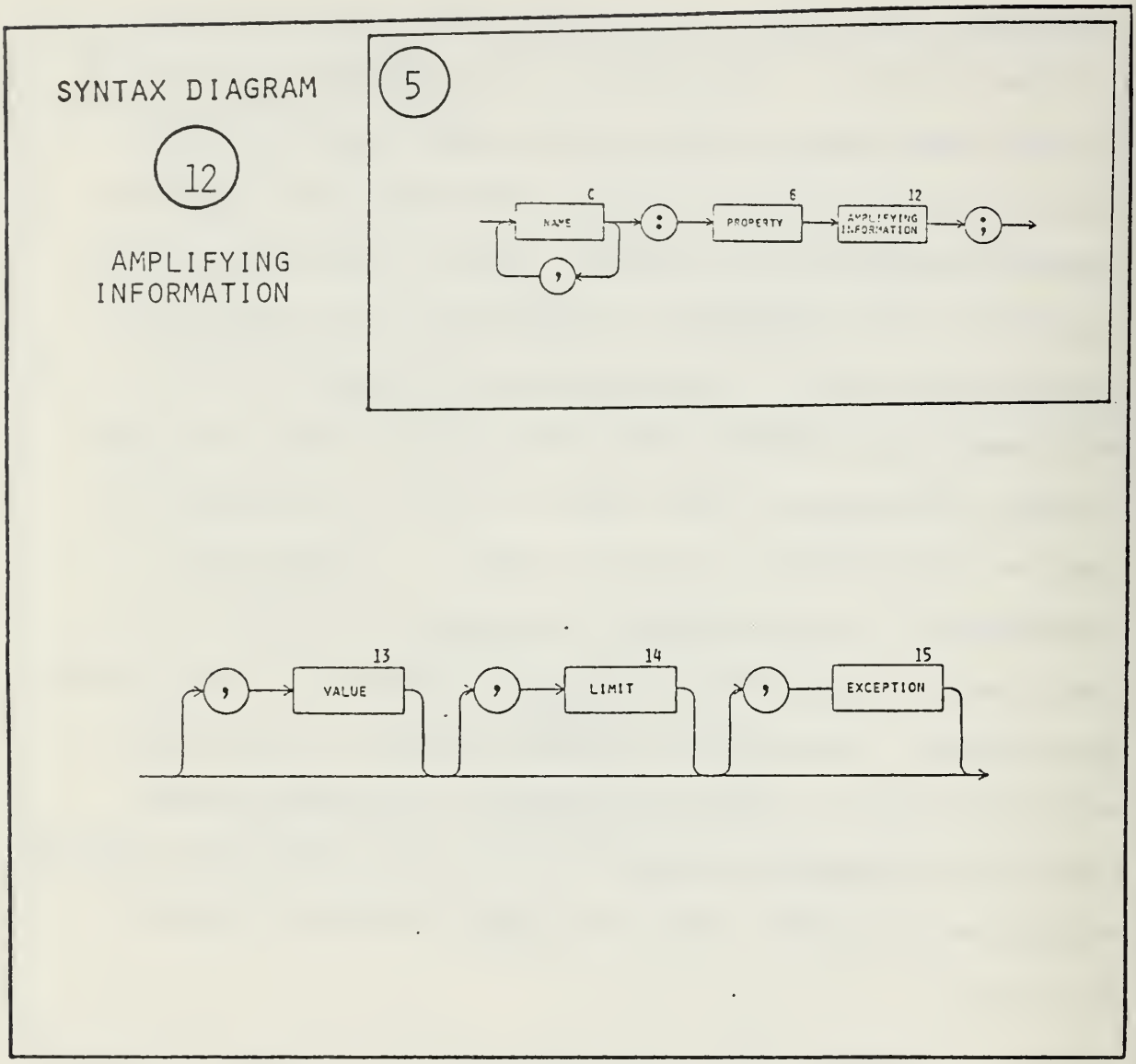


Figure 4-26

Example IV - 21: Sample of data descriptions using amplifying information

```
A, B: BOOLEAN, INITIAL VALUE (0), LIMIT (0,1), ERROR (1);
VOLTAGE: REAL, LIMIT (-5.0..+5.0), VOLTAGE←RESET;
SALUTATION: ARRAY [5] OF CHARACTER, CONSTANT VALUE ('H',
  'E','L','L','O');
```

e. Amplifying Information

Amplifying information is optional information which is associated with data names. As shown in Syntax Diagram 12 and Example IV - 21, it consists of initial or constant values of data items, bounds or limits on the valid values of data items and the designation of an exception handling process or operation. The syntactic structure and semantics of each of the elements of amplifying information are discussed in detail in the following sections along with a discussion of the appropriate parts of Example IV - 21.

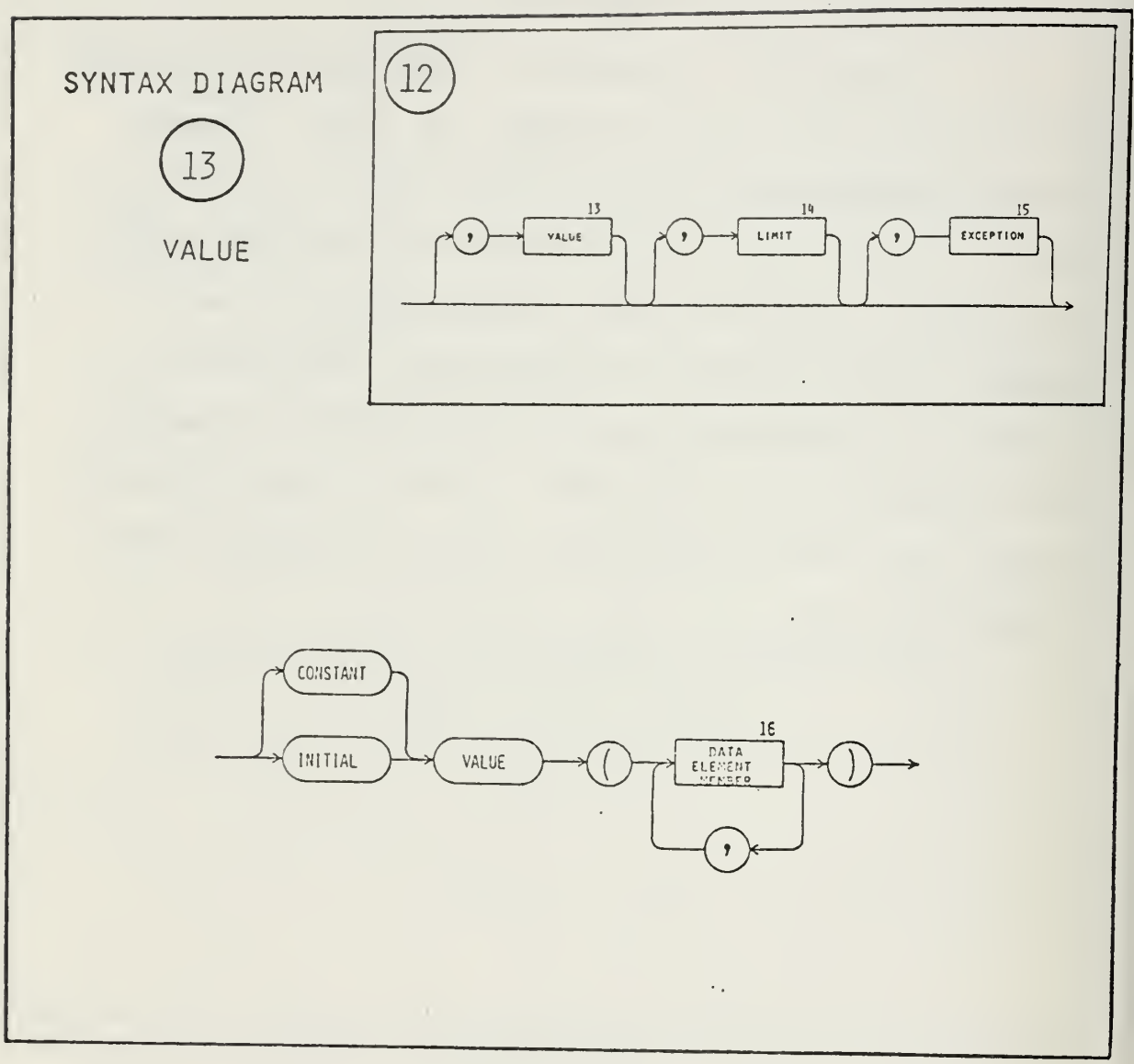


Figure 4-27

Example IV - 22: Sample of value portion of amplifying information

```
INITIAL VALUE (0)
CONSTANT VALUE ('H','E','L','L','O')
```

(1) Data Value Information. Initial value information as presented in the first example of Examples IV - 21 and IV - 22 is used for assignment of an initial

value to a data name upon every entry into a process. The value is not protected from change during process execution.

Constant value information which is presented in the last lines of Example IV - 21 and IV - 22 is used for assignment of a constant value to a data name. The value assigned is the only value that the name can be associated with. The data name becomes a protected ("read only") name and an attempt to assign a value to this name in the operation section will result in an error.

The data element member is described in Section IV.C.4.e.(4). It is a value which is assigned to a data name. This value must be a valid member of a set which is defined in the property definition of the data element. A sequence of data element members separated by commas is used to assign values sequentially to the elements of arrays, lists or sets. The last lines in Examples IV - 21 and IV - 22 provide such an example of multiple assignments.

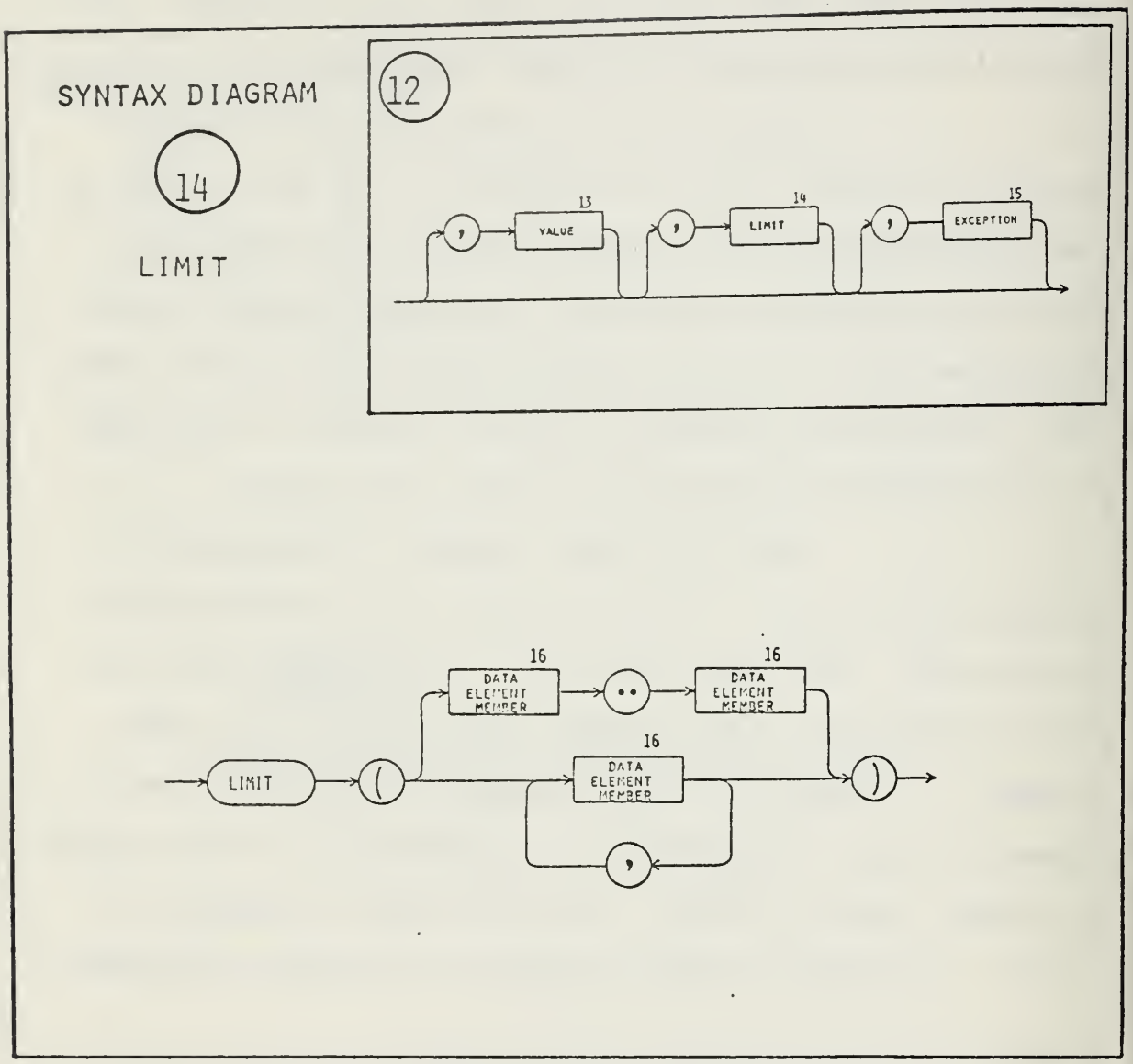


Figure 4-28

Example IV - 23: Sample limit portion of amplifying informatio

```
LIMIT (0,1)
LIMIT (-5..+5)
```

(2) Data Value Limit Information. The limit information provides a vehicle for specifying valid values which can be assigned to a data name. These limits may be specified as a list of data element members

(Section IV.C.4.e.(4)) separated by commas as shown in the first line of Examples IV - 21 and IV - 23. Limits may also be specified as bounds on the valid data members as in the second line of Examples IV - 21 and IV - 23. Bounds are specified by indicating a lower limit followed by two contiguous periods then an upper limit. The lower limit must be less than or sequentially preceding (in the case of a set) the upper limit. Only one pair of bounds is allowed.

All data names which have limits associated with them are automatically checked for valid values during process execution. If the limits are exceeded, then either the specified exception process or a system default exception operation is performed. This feature reduces the amount of error checking the user must do in the operation section and permits the user to concentrate on problem solving rather than error checking in the operation section.

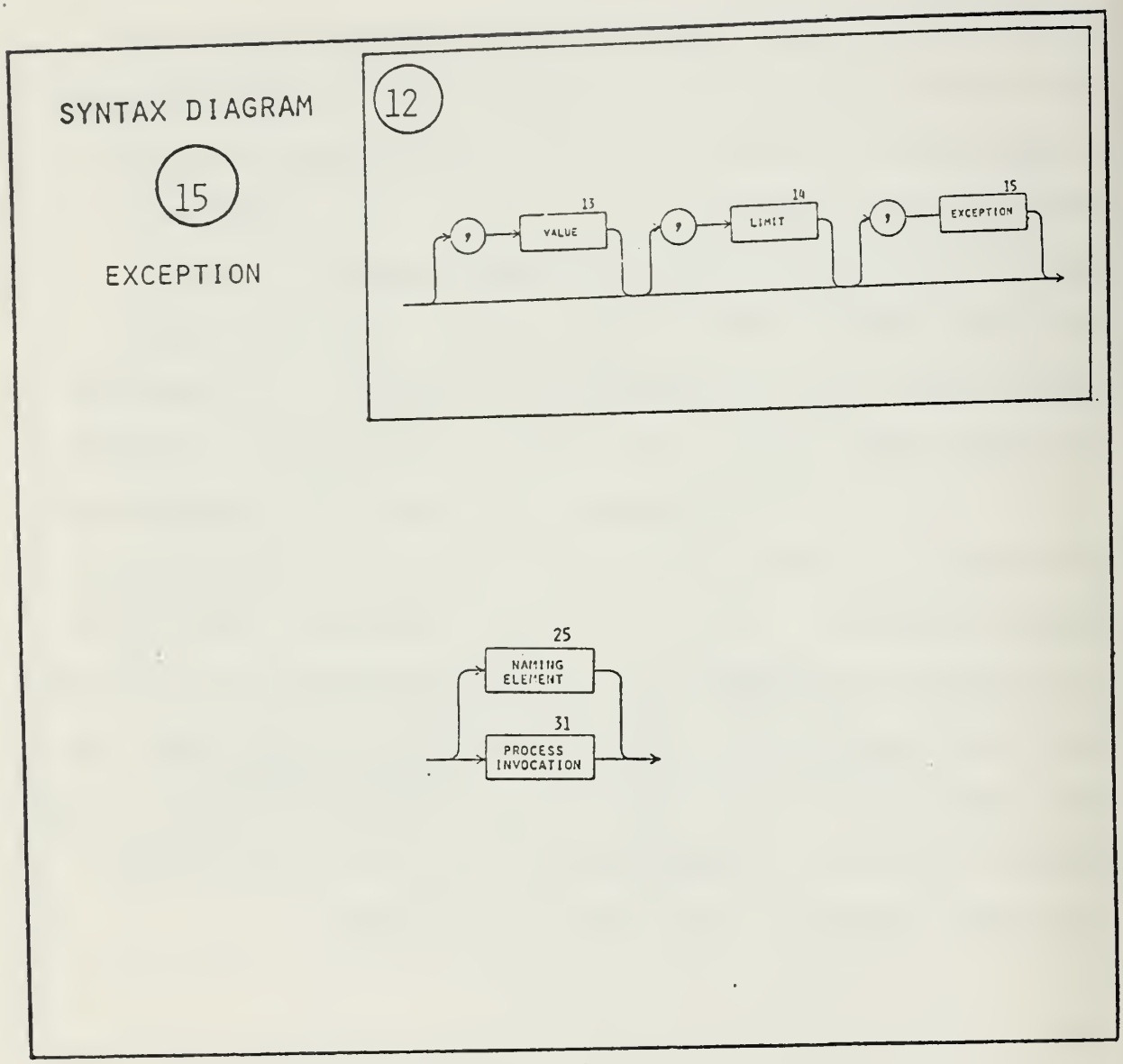


Figure 4-29

Example IV - 24: Sample exception portion of amplifying information

```
ERROR (1)
VOLTAGE←RESET
```

(3) Exception Handling Information. The exception portion of amplifying information provides a method of specifying operations to be performed when a specified limit has been exceeded during execution of a process. The

exception allows the possibility of programmatic correction of an out of limits condition and a return to execution. This procedure may be done through the use of a naming element (Section IV.C.5.c.) and is shown in the second line of Example IV - 21 and IV - 24. Alternatively, this mechanism can also allow actions such as the printing of an error message and continuation of execution or termination of execution. The first line in Examples IV - 21 and IV - 24 show this case. Unless the exception handling process has a STOP operation in it (Section IV.C.5.b.(4)), the exception handling process will return to the next succeeding machine instruction after the instruction in which the out of limits condition was detected. If no exception operation is specified, a default system exception handling process will be executed. The default process should be a run-time system monitor process which should indicate the error and its location and then terminate the execution of all processes.

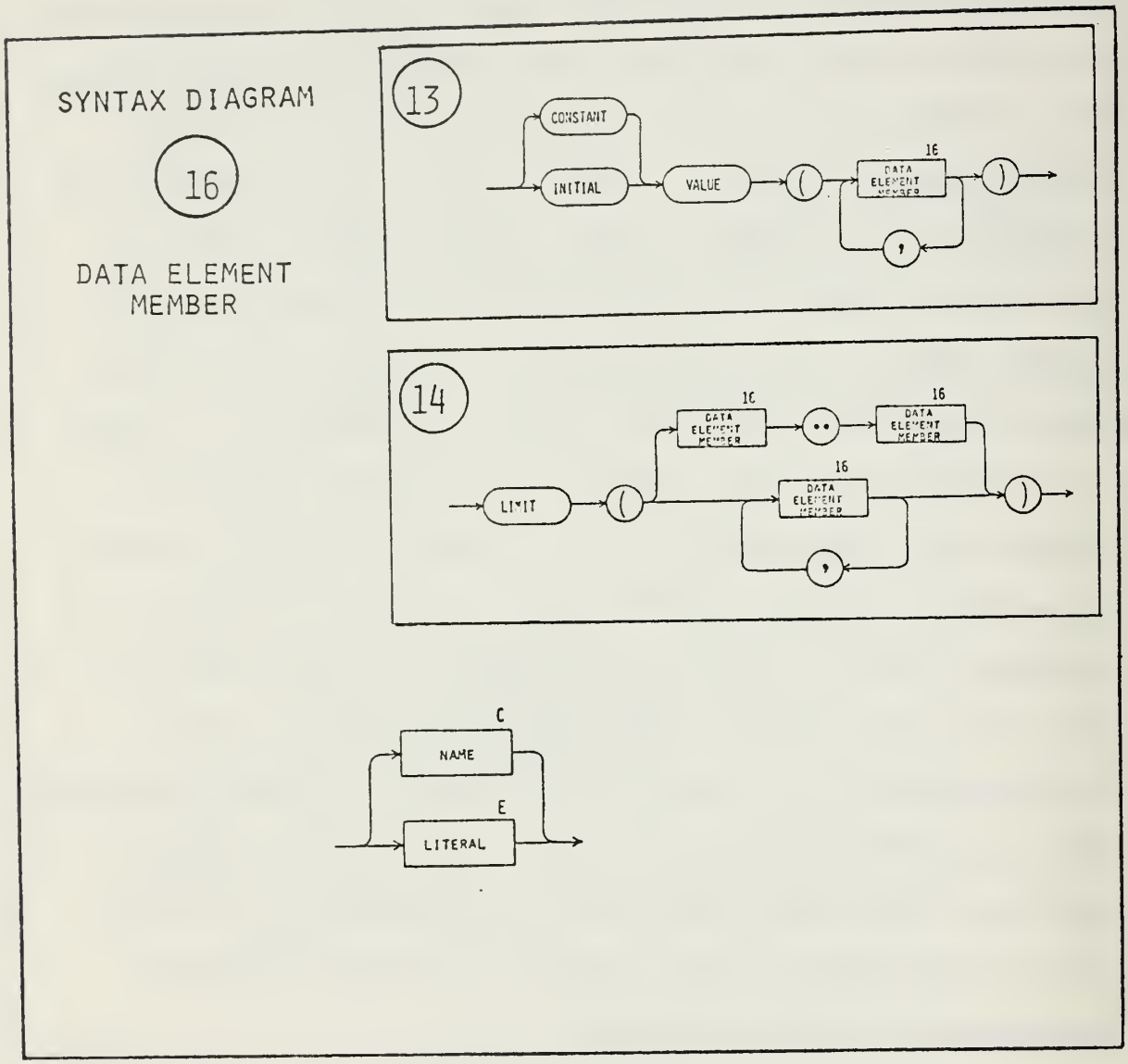


Figure 4-30

(4) Data Element Member. Data element members are members of the set of valid data elements. Data elements are specified in the property section of the associated data name. For example, the data element ALPHA could have as valid members the set of alphabetic characters A through Z. A valid data element member of ALPHA would then be any one of the characters in the specified set.

The literal element of this syntactic category is described in Section IV.C.7.c.(3) and the name element is described in Section IV.C.7.c.(1).

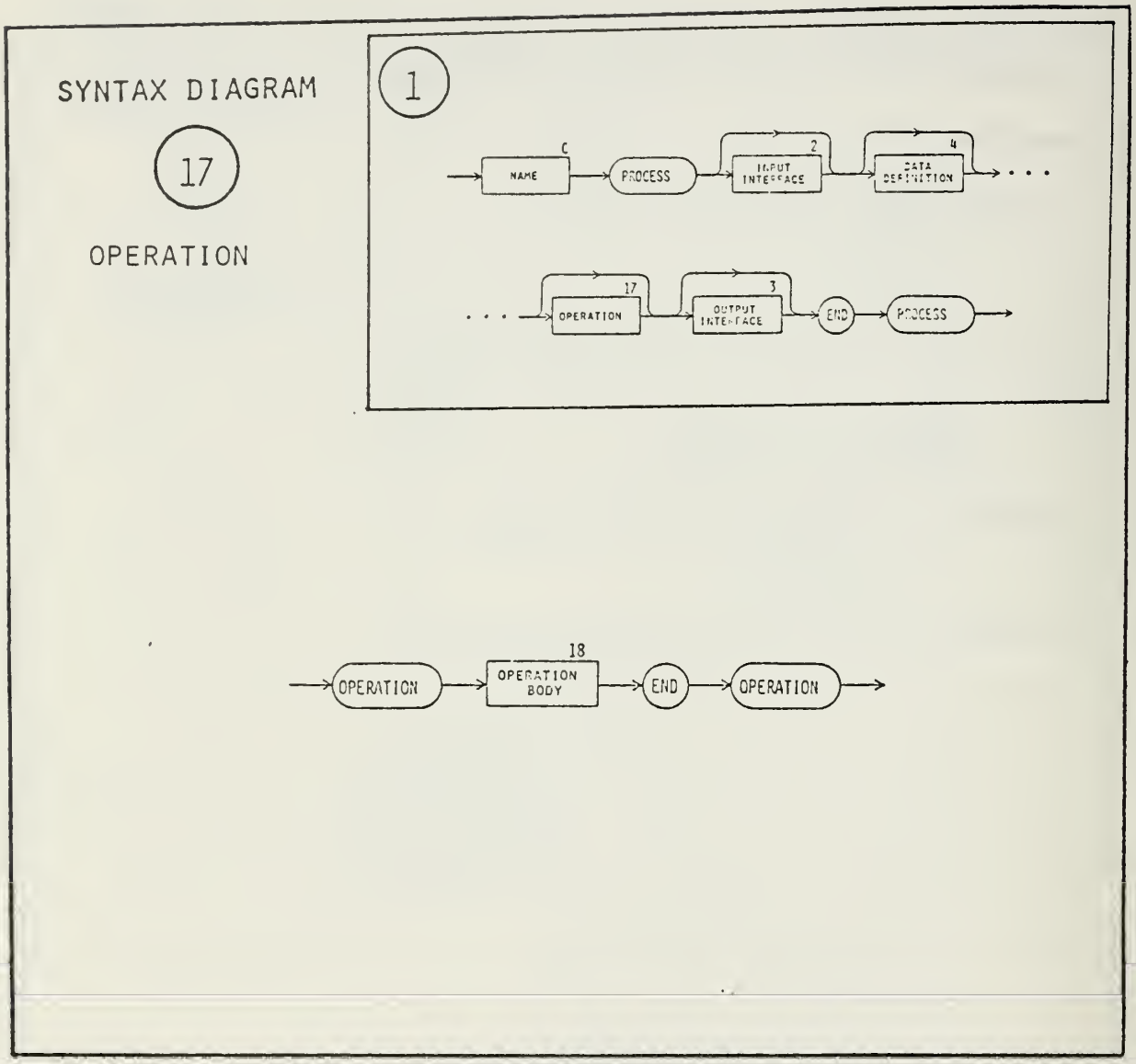


Figure 4-31

Example IV - 25: Sample operation section

```
OPERATION
(A>=B): DEPART;
A←B;
END OPERATION
```

5. Operations

a. General Structure

The operation section is the active component of a process. In the operation section local process data

is created, modified or destroyed by the base language operation elements or other existing processes. It is through the actions of the operation section that the desired function of the process is computed.

The structure of the operation section is shown in Syntax Diagram 17 and in Example IV - 25. The operation body which is described in subsequent sections is separated from the other sections of the process by the reserved words OPERATION and END OPERATION.

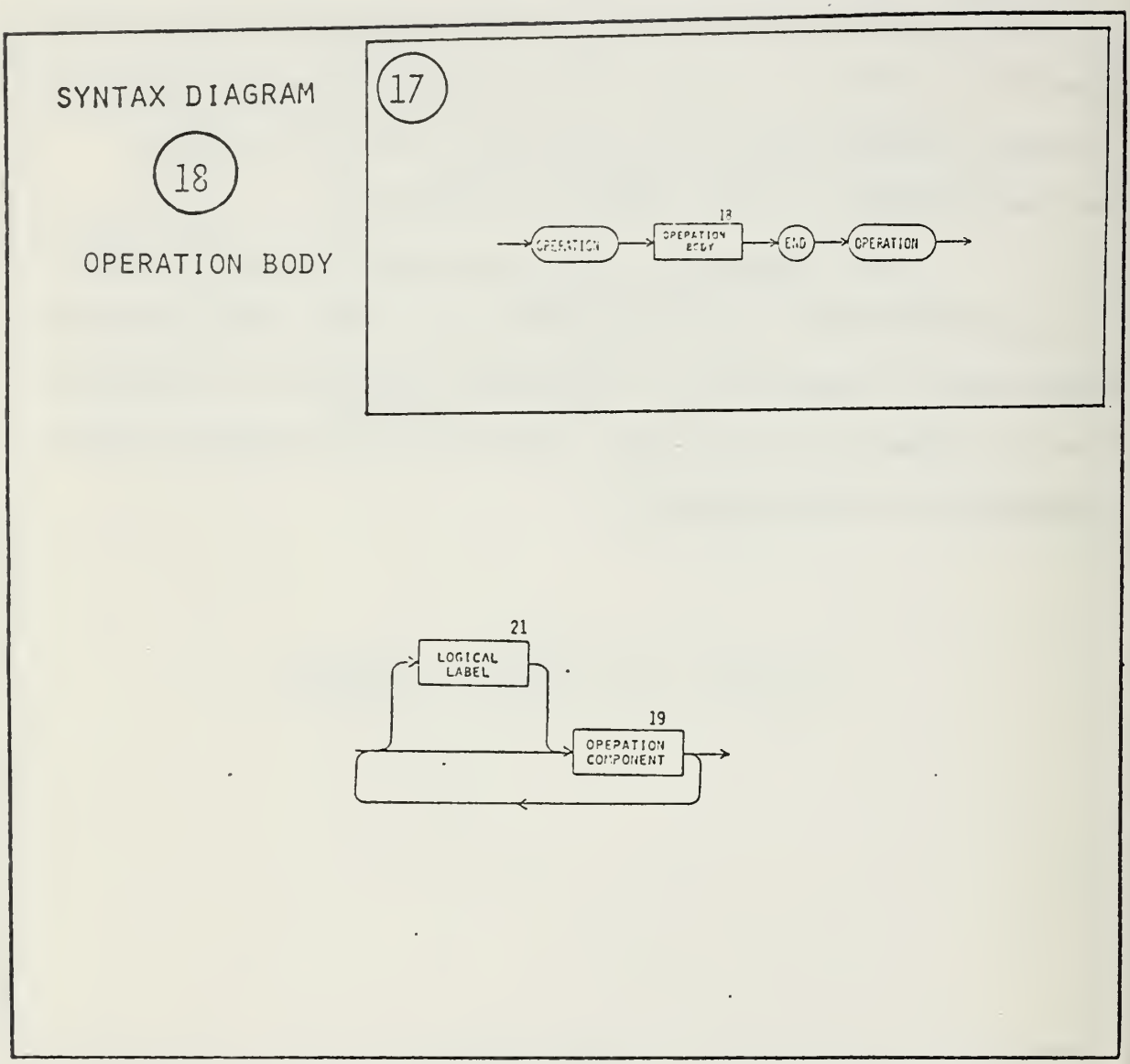


Figure 4-32

Example IV - 26: Sample operation body

```
(A>=B): DEPART;
A←B;
```

(1) Operation Body. The operation body is a physical subunit of an operation and consists of a sequence of one or more operation components (Section IV.C.5.a.(2)) which may be optionally preceded by a logical label

(Section IV.C.5.b.(1)). A logical label contains a logical expression that allows execution of the immediately succeeding operation component when the expression evaluates to the boolean value "true". Sample operation components and logical labels are shown in Example IV - 26. The syntax and semantics of these language constructs are explained in Sections IV.C.5.b. through IV.C.5.d.

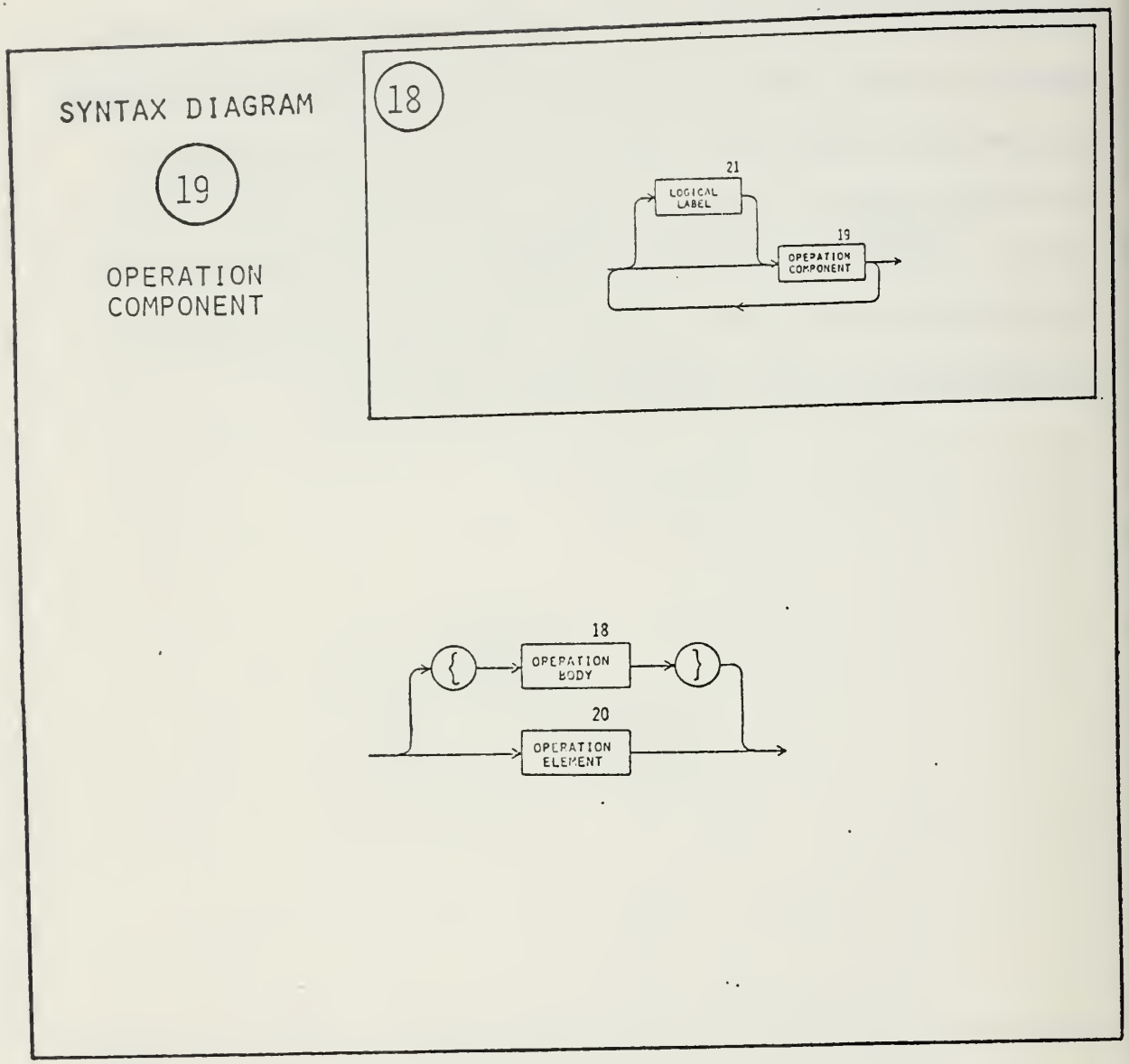


Figure 4-33

(2) Operation Component. An operation component is either a single operation element which is described in Section IV.C.5.a.(3) or an operation body enclosed within a left-right curly brace pair ({ }). This enclosed operation body, in effect, forms a group of operation elements which may be considered to be one operation element. An operation element is analogous to a statement and the

left-right curly brace pair is analogous to the BEGIN-END pair in the language PASCAL.

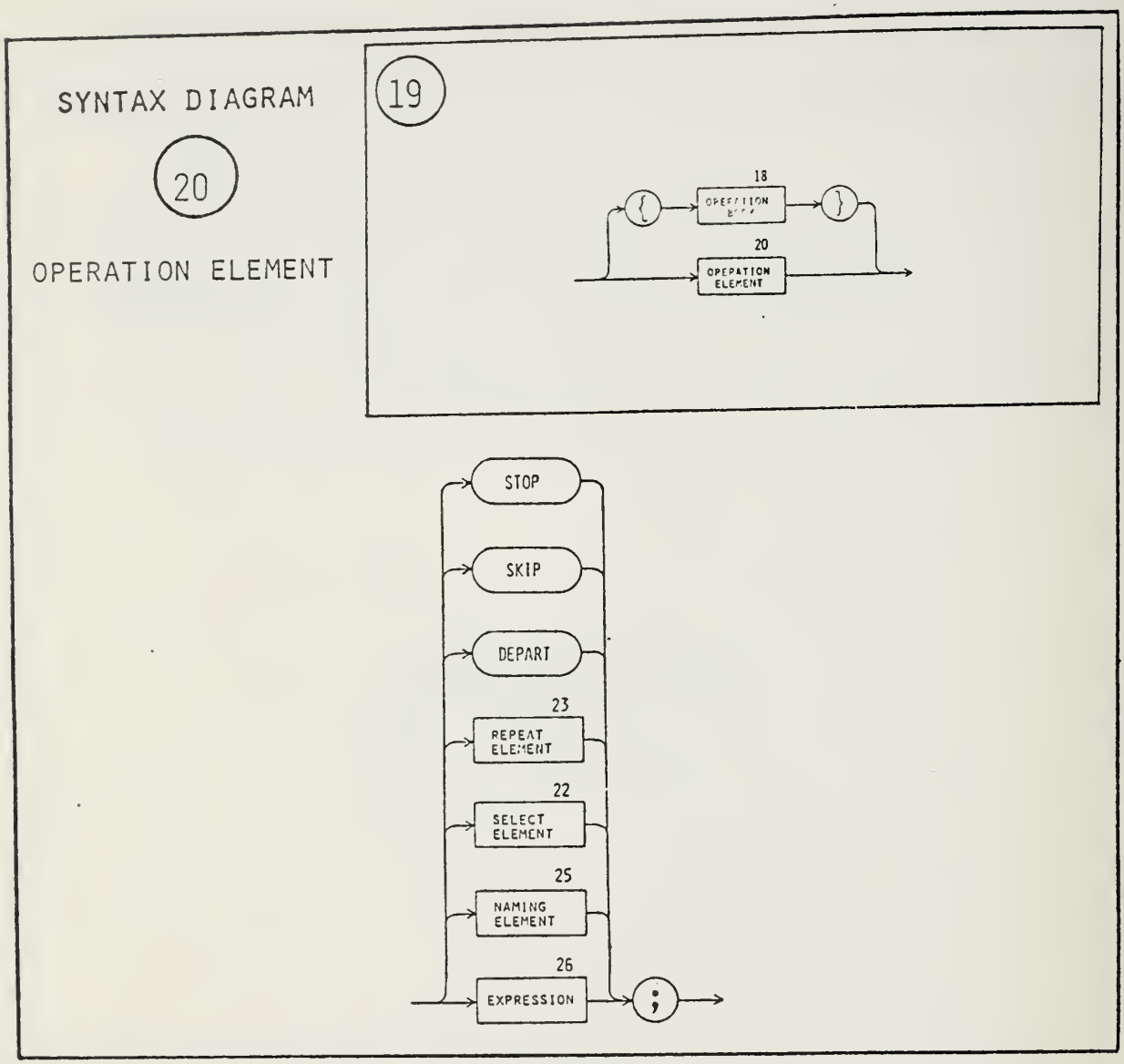


Figure 4-34

(3) Operation Element. The operation element is the primary unit of the operation section. Syntax Diagram 20 shows all the operation elements of the base language. These elements along with the logical lable provide process execution control. These control structures include: iteration, selection, and data manipulation. Since each operation element is an individual unit which

performs a specific function, they are terminated by a semicolon to separate one from the other within an operation section. Each of the operation elements shown in the above diagram will be discussed in the succeeding sections.

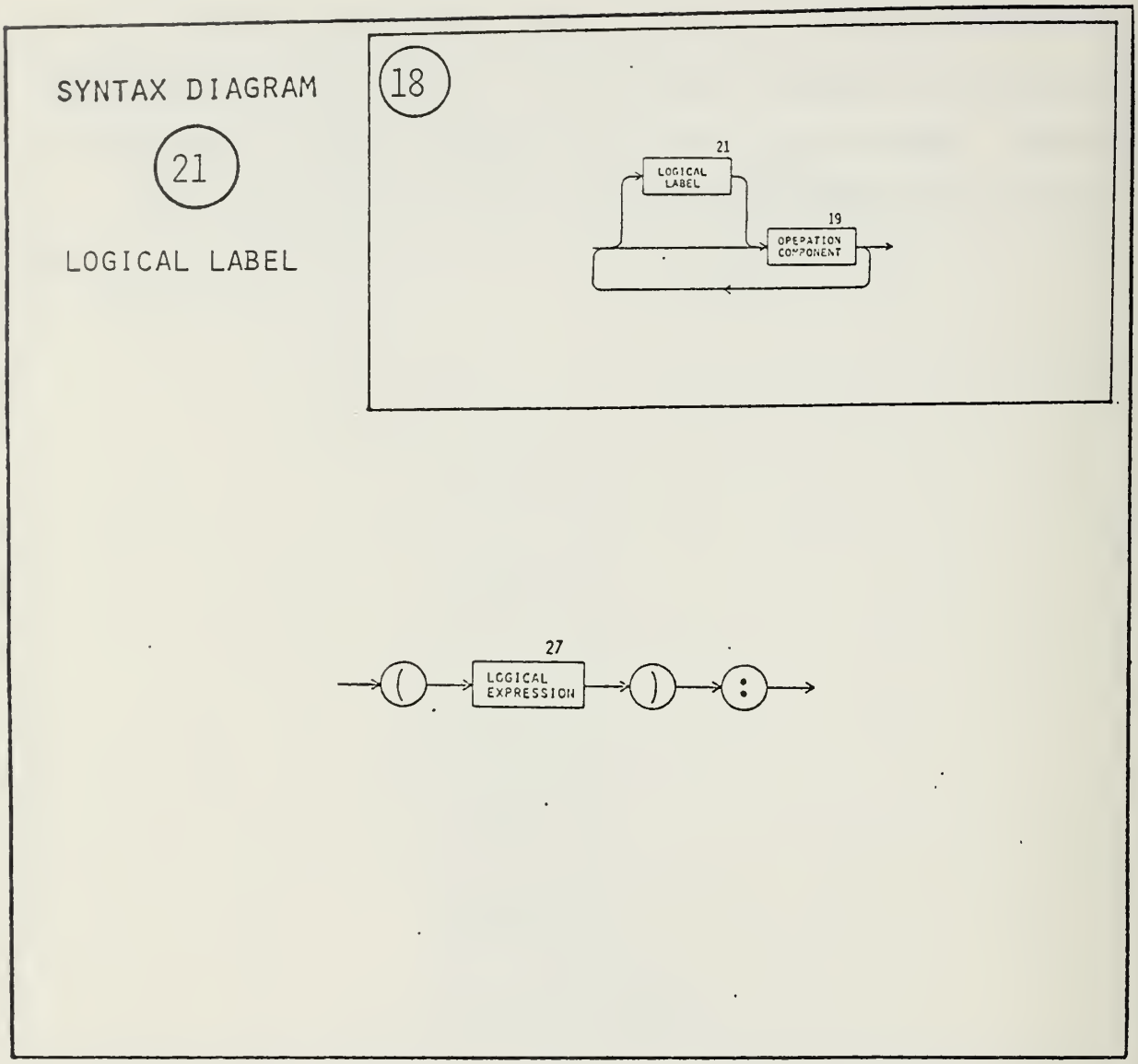


Figure 4-35

Example IV - 27: Sample logical label

```
(A>=B): DEPART;
A←B;
```

b. Control Structures

(1) Logical Label. The logical label is a conditional element which is analogous to the IF-THEN construct in many computer languages. The logical label

consists of a logical or relational expression (Sections IV.C.5.d.(1) and IV.C.5.d.(2)) contained between a left-right parenthesis pair and followed by a colon as shown in Syntax Diagram 21. During execution, the expression in the label is evaluated. A "true" result allows execution of the operation element which immediately follows the logical label. A "false" result causes the element following the label to be skipped. The logical label is not a "go-to" type label. It does not indicate a point to which program execution is transferred. A logical label is evaluated in the normal sequence of program execution and acts as a binary switch to control branching.

Example IV - 27 shows an application of the logical label. If the expression (A>=B) is true then the DEPART element (Section IV.C.5.b.(6)) is executed. Otherwise, the DEPART element is bypassed and the next element, A←B; is executed. Example IV - 1 shows an application of a logical label in a select element (Section IV.C.5.b.(2) which acts as an if-then-else construct. A select element is used in conunction with the logical lable in this case because the value of the data name A used in the label is modified by the element A←A-B, which follows the label. If there is no modification of the data names in the operation component following the label which are also used in a logical label as test variables, a simpler construct can be used to represent the if-then-else. This construct is shown in Example IV - 28.

Example IV - 28: Sample IF-THEN-ELSE construct using logical labels

```
(A<=10): #PERFORM THIS OPERATION#;
(A>10): #PERFORM THIS OPERATION#;
```

This type of construct is much clearer than an if-then-else construct. All the conditions which are to be met for the performance of any particular operation precede the operation to be performed. This makes the conditions highly localized and visible.

THIS PAGE INTENTIONALLY LEFT BLANK

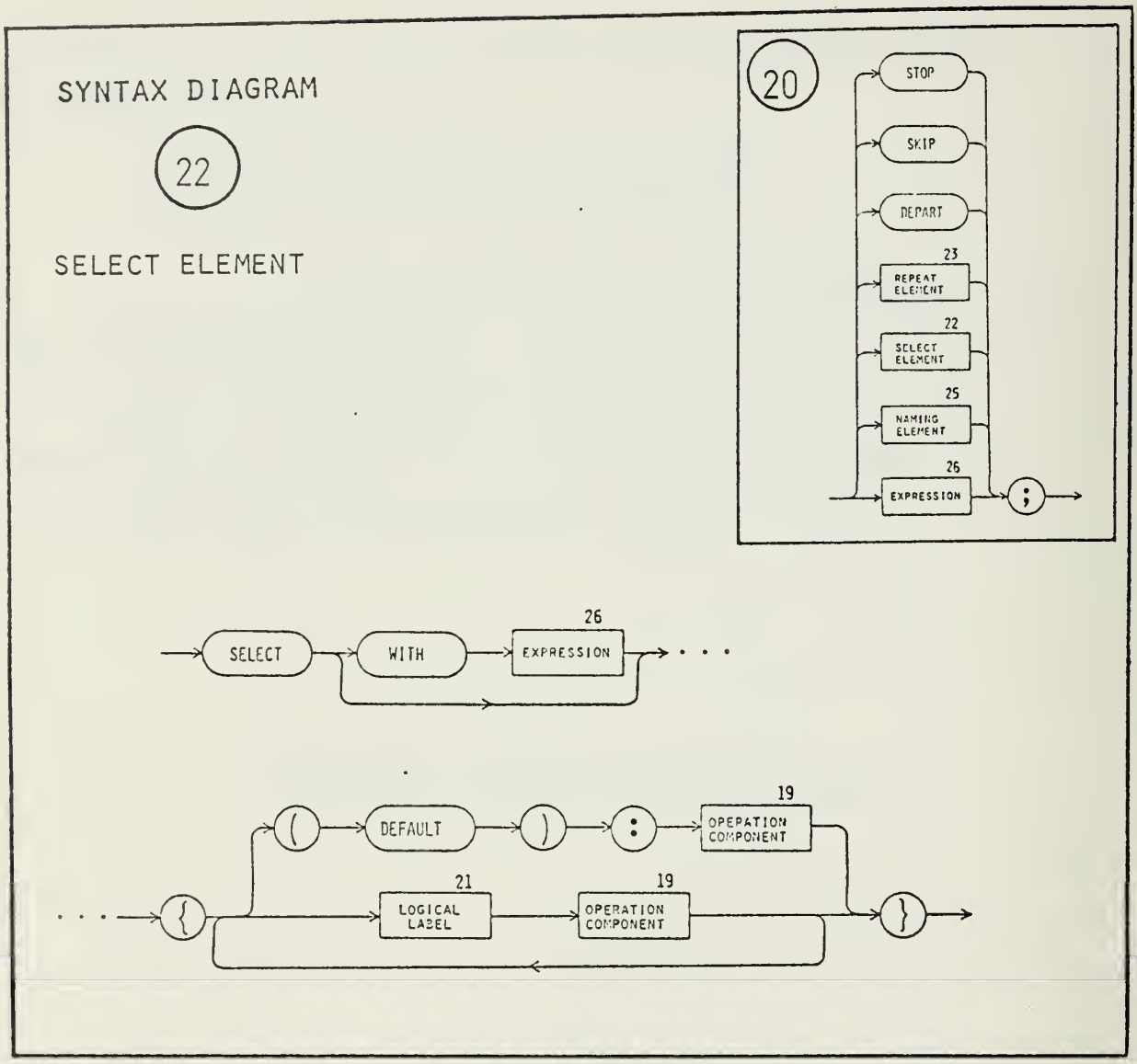


Figure 4-36

Example IV - 29: Sample select element

```
SELECT WITH (A + B)
{
     (@ = 1): X←X+1;
     (@ = -5): Y←Y+1;
     (@ = 0): Z←Z+1;
     (DEFAULT): {M←M+1;
                 X,Y,Z←0;
                }
};
```

(2) Select Element. The select element allows selection of one operation component for execution from a set of operation components. It is similar to the case statement in many other computer languages. When using the select element, only one of the possible operation components will be executed. The remaining select element components will be bypassed. The logical label which precedes each operation component in the set is used to determine which component is selected and executed. The syntactic structure shown in Syntax Diagram 22 allows an optional expression clause to be used in the selection of the proper operation component. This option is shown in Example IV - 29. Upon execution of the select element, the expression is evaluated and the answer, indicated by the symbol @ (Section IV.C.5.d.(4)(d)), may be used in a logical label to cause selection of the proper component. An optional DEFAULT may also be used in a label for the last operation component in the select set. If the DEFAULT is used and none of the logical labels evaluate to true, the DEFAULT component is executed. In Example IV - 29 if the answer of (A + B) is not 1, -5 or 0 the DEFAULT operation is executed, M is incremented by 1 and X, Y and Z are assigned 0. If the DEFAULT label is not used and none of the logical labels evaluate to true then none of the operation components are executed and an exit from the select element occurs.

Example IV - 1 shows an application of the select element without the expression clause. This select construct operates the same way as the one described above except that the only expressions evaluated are the logical expressions in the logical labels.

In the event that more than one logical label in a select element could evaluate to true, the component associated with the first logical label, in sequence, which evaluates to true is the only component executed. All other components in the set are skipped.

THIS PAGE INTENTIONALLY LEFT BLANK

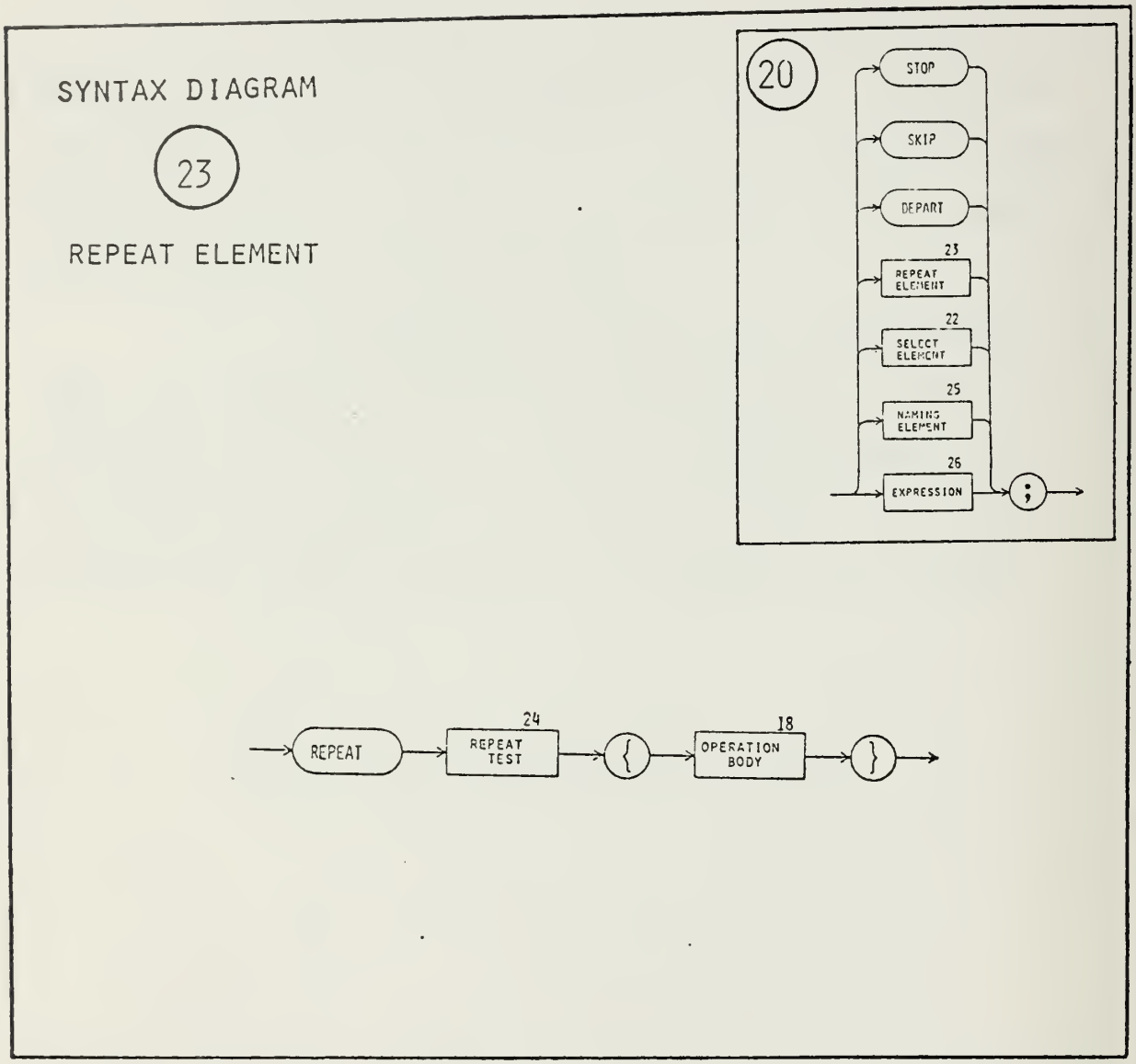


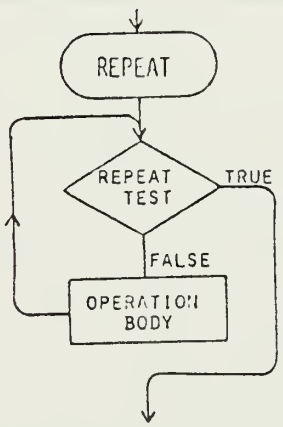
Figure 4-37

Example IV - 30: Sample of a simple repeat element

```
# ASSUME THAT THE VALUE OF B IS 10
# AND THAT THE VALUE OF A IS 0
# AT THE START OF EXECUTION.

REPEAT (A>B): TERMINATE
{
   A←A+1;
}
```

(3) Repeat Element. The repeat element provides a control structure for iteration in the base language. It incorporates all standard loop constructs, the do while, do for, and repeat until or similar constructs which may be found in many modern computer languages. The syntactic structure of the repeat element is shown in Syntax Diagram 23. It consists of the reserved word REPEAT followed by a repeat test. The repeat test controls the number of times the succeeding operation body (Section IV.C.5.a.(1)) is executed. The repeat test condition is evaluated prior to execution of the operation body. Figure 4-38 presents a simple flow diagram which indicates the flow of control in the repeat element. The repeat test contains all testable conditions which are pertinent to the execution of the



REPEAT ELEMENT CONTROL FLOW DIAGRAM

FIGURE 4-38

repeat element. Execution of a repeat element can be terminated only by meeting one of the test conditions in the repeat test.

In Example IV - 30 the repeat element shown will execute until A is greater than B. Assuming the initial conditions stated in the example, A will be incremented in the operation body until the value of A is greater than the value of B. In this case, the operation body will be executed 11 times. When the repeat test condition (A>B) is satisfied, the execution of the repeat element will be terminated. Process execution continues with the next operation component. If the value of A is greater than the value of B when the repeat element is initially executed, the repeat test condition will be met, the operation body will not be executed and process execution will skip to the next operation component.

THIS PAGE INTENTIONALLY LEFT BLANK

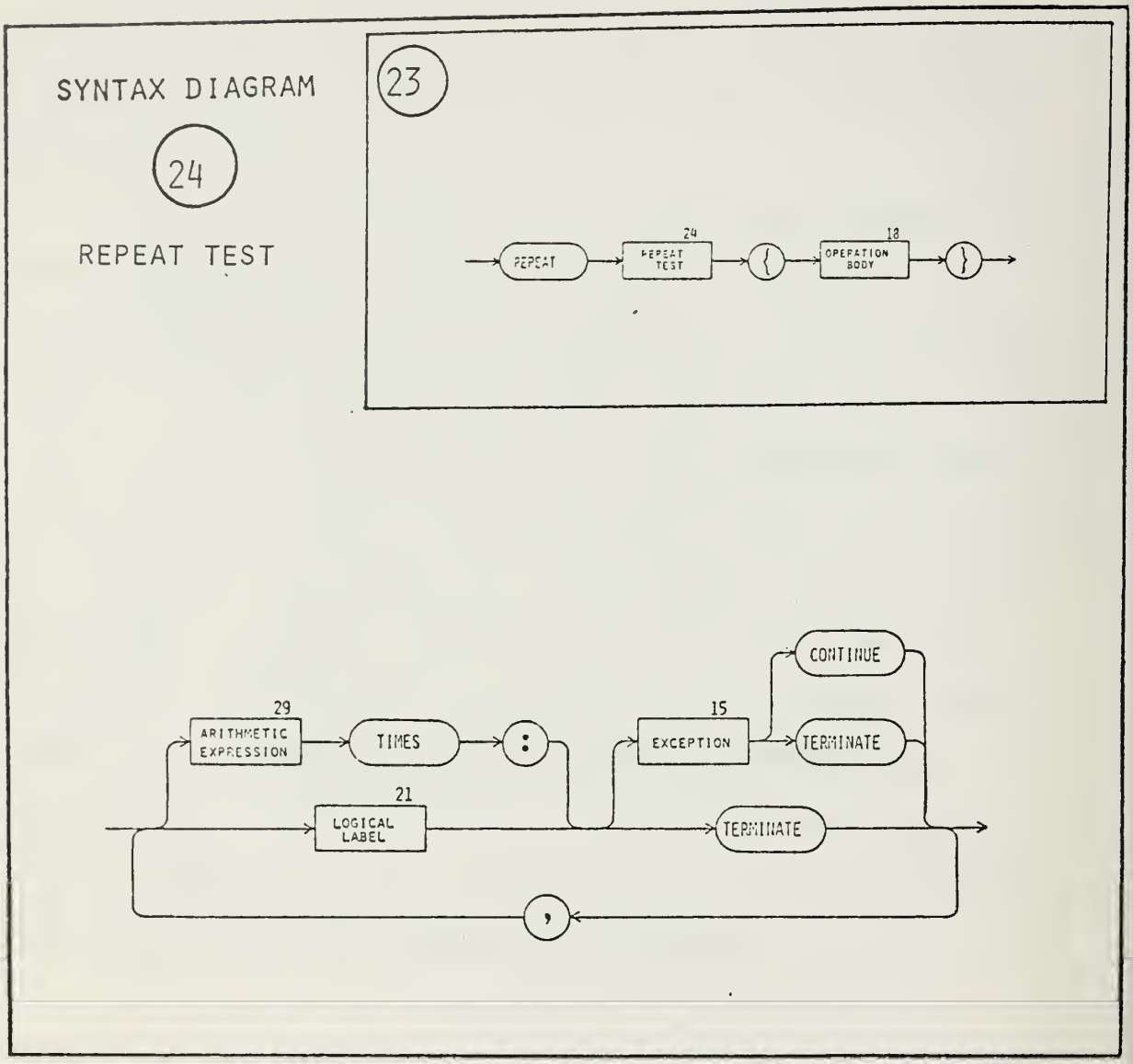


Figure 4-39

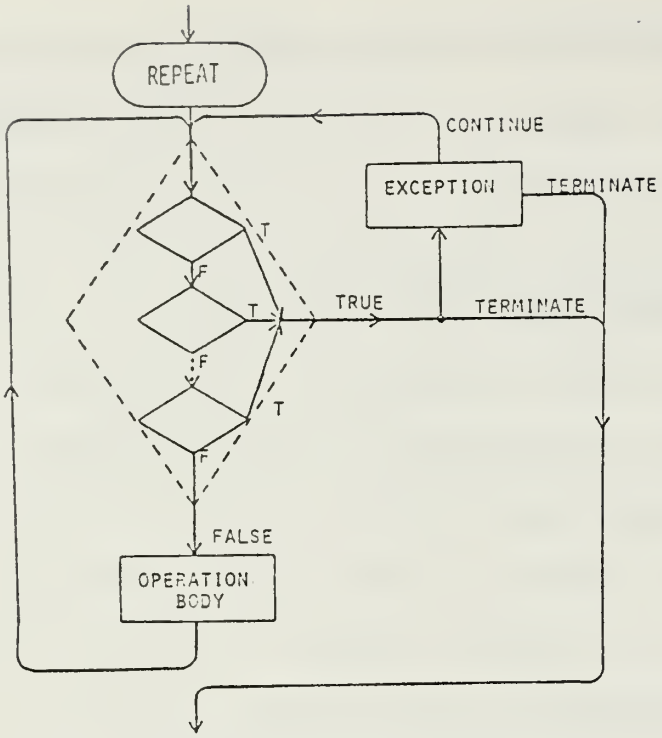
Example IV - 31: Sample repeat test component of a repeat element.

```
REPEAT (A>B): TERMINATE,
100 TIMES: ERRORMSG TERMINATE,
(A=B): A←SPECIALPROC(A) CONTINUE
{
 # OPERATION BODY
};
```

The repeat test consists of test conditions and loop control actions which are performed when these test conditions are met. A test condition may be a logical

label or a statement of a finite number of execution iterations. The number of iterations in an iteration statement is represented by an arithmetic expression (Section IV.C.5.d.(3)) followed by the reserved word TIMES and a colon symbol (:). Upon entry into the repeat element, the arithmetic expression is evaluated and its value is used to initialize an iteration counter. The iteration statement may be used to guarantee loop termination since the iteration counter is not accessible to the user. This feature may be useful in process debugging or in real time processing applications. A loop control action may be the reserved word TERMINATE or a user defined exception (Section IV.C.4.e.(3)) followed by the reserved words TERMINATE or CONTINUE. The TERMINATE action causes the termination of the repeat element when the associated condition is met. If the reserved word TERMINATE follows an exception process or operation, termination occurs after the exception process or operation is performed. The CONTINUE action which may be used only with a user defined exception allows execution of the repeat element to continue after the intermediate exception process or operation is performed.

A detailed diagram of repeat process control flow is depicted in Figure 4-40. As the diagram indicates, each condition in the repeat test is evaluated sequentially. If the condition evaluates to true, the associated action is



DETAILED REPEAT ELEMENT CONTROL FLOW DIAGRAM

FIGURE 4-40

performed. Otherwise, the next sequential test condition is evaluated. If none of the test conditions evaluate to true, the repeat operation body is executed. Upon completion of the repeat operation body, execution control is transferred to the repeat test and the next iteration begins. An exception process or operation associated with a test condition is executed prior to the terminate or continue action.

Example IV - 31 provides a sample of each of the repeat test constructs discussed. The first condition evaluated is A>B. If the expression evaluates to true, the repeat element is terminated. Otherwise, the next

condition 100 TIMES is evaluated, that is, the iteration counter is tested to determine if 100 iterations have been performed. If 100 iterations have been performed the ERRORMSG process is executed and then the execution of the repeat element is terminated. Otherwise, the last condition is evaluated. If A equals B then the SPECIALPROC process is executed, the result is assigned to A and the execution of the repeat element continues. Otherwise, the repeat operation body is executed.

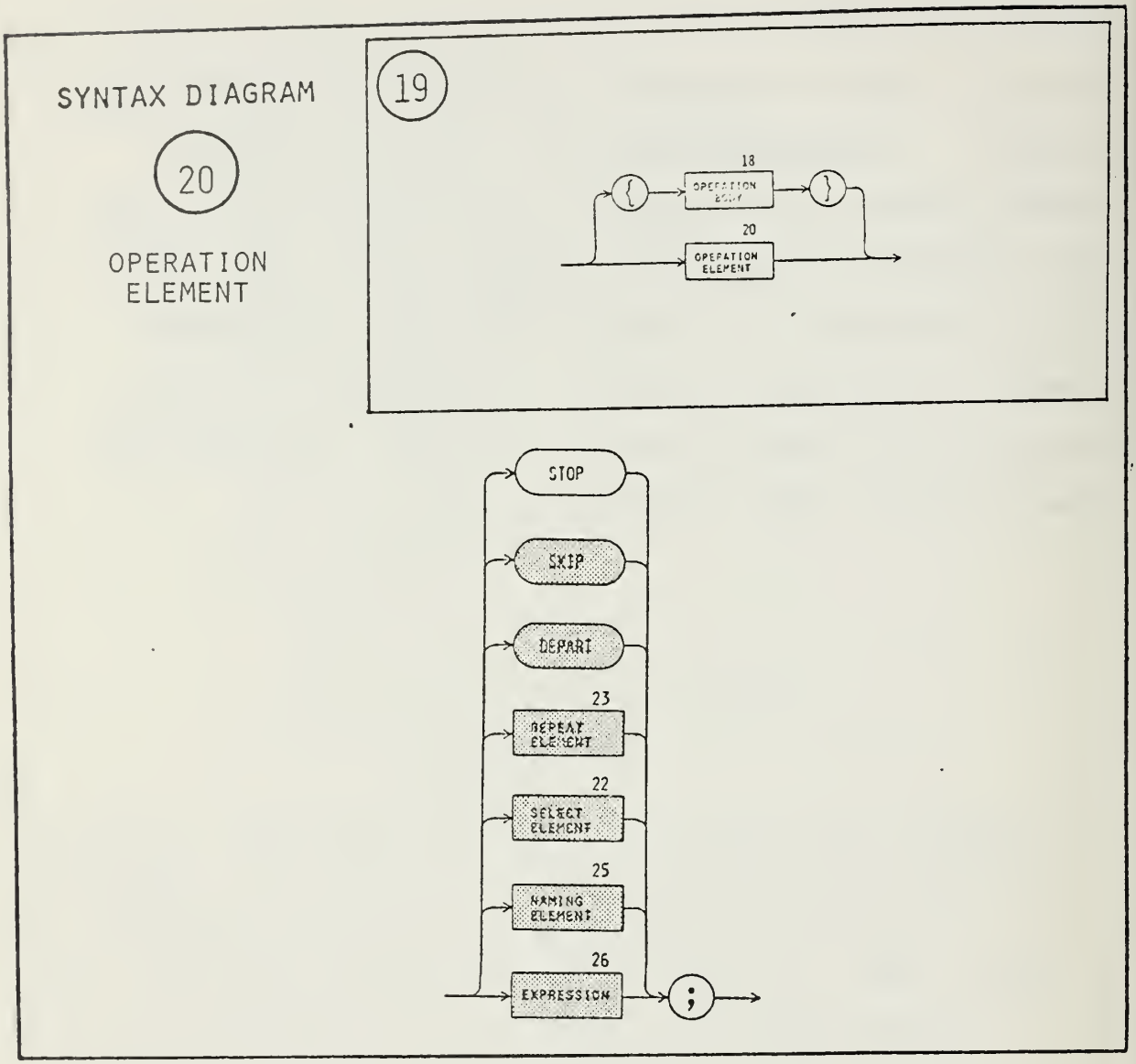


Figure 4-41

(4) Stop Element. The stop element is the reserved word STOP. A STOP can be used at any point in a process. It is analogous to a machine halt in assembly languages and causes execution of the process to halt at the point where the STOP was executed. This operation is useful in exception processing and in process debugging.

THIS PAGE INTENTIONALLY LEFT BLANK

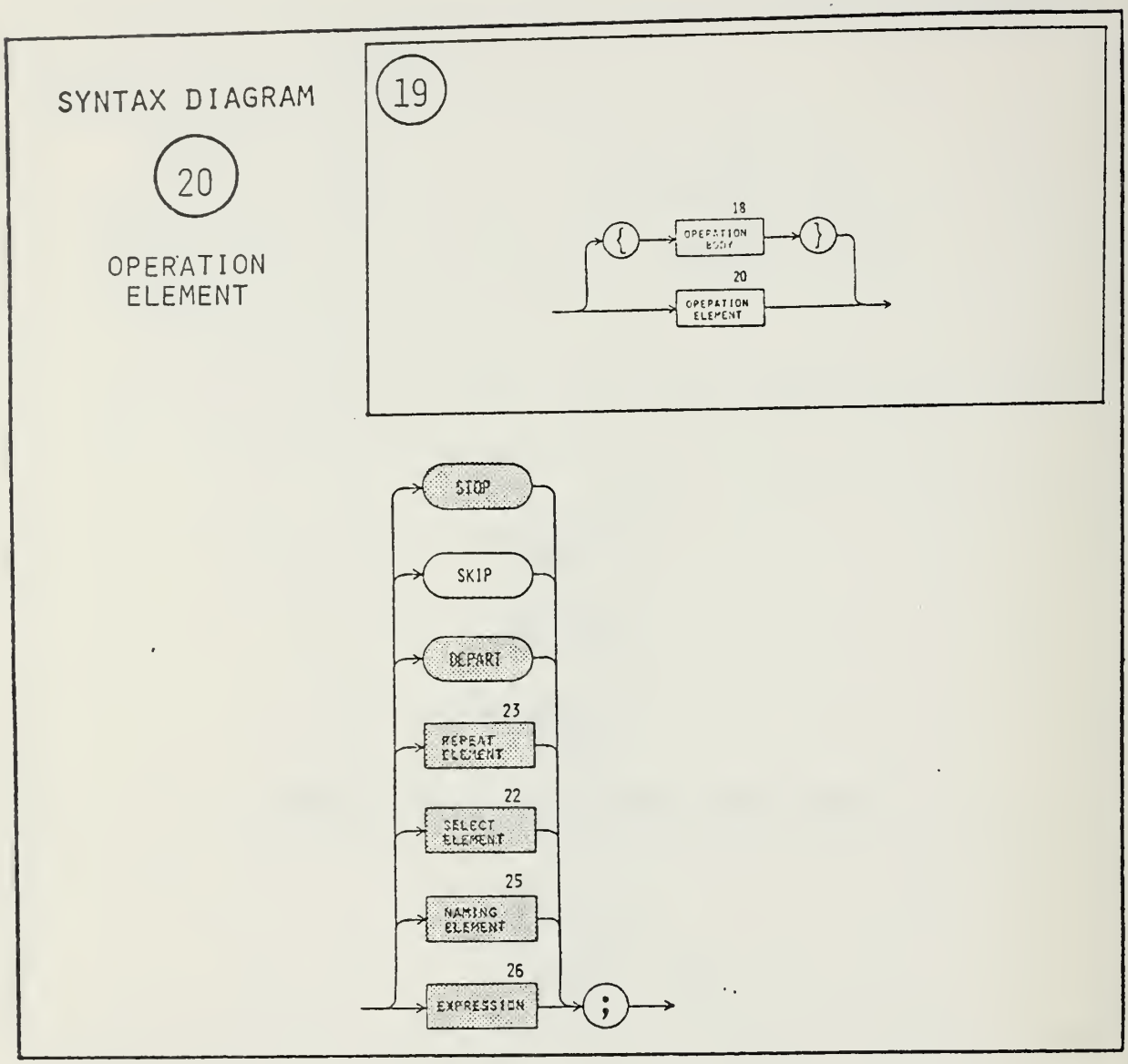


Figure 4-42

(5) Skip Element. The skip element is the reserved word SKIP. The skip element causes process execution to transfer unconditionally from the point at which the reserved word SKIP is encountered to the beginning of the next sequential operation component. When a SKIP is used in a sequence of individual operation elements it causes essentially no operation to be performed. However,

when used within a group of operation elements enclosed within a left-right curly bracket pair ({ }) it causes execution to skip to the next operation component after the group. A SKIP element is invalid within a REPEAT element.

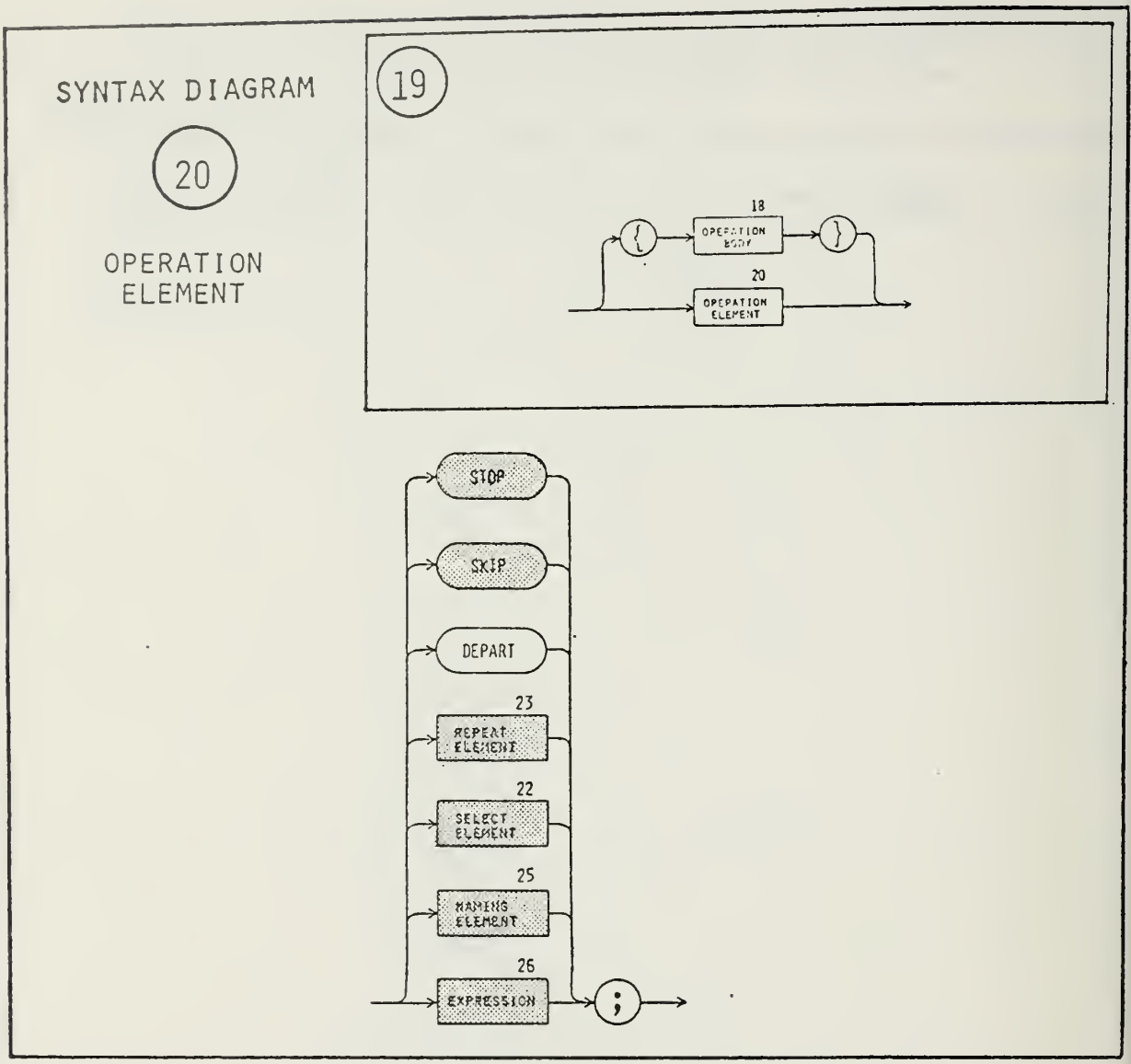


Figure 4-43

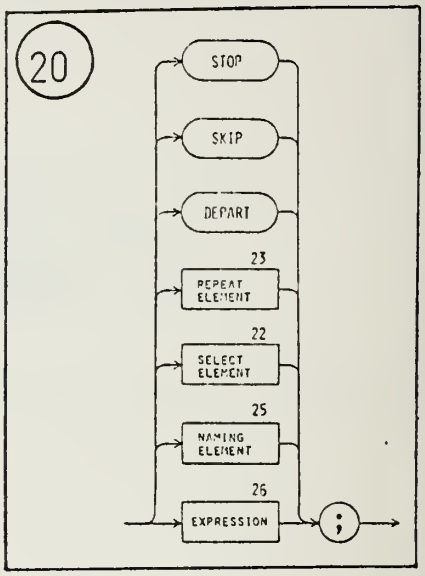
(6) Depart Element. The depart element is the reserved word DEPART. When encountered during execution it causes process execution to immediately transfer out of the operation section to the output interface section. It may be used to avoid execution of a portion of a process based on some condition. The effect is the immediate return of execution to the next higher level process. The DEPART is invalid within a REPEAT element.

THIS PAGE INTENTIONALLY LEFT BLANK

SYNTAX DIAGRAM

25

NAMING ELEMENT



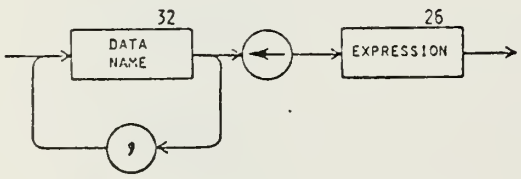


Figure 4-44

Example IV - 32: Sample naming elements

```
A←B+C;
Y←SIN(X)
BUF[A]←Y;
A,B,C←0;
```

c. Naming Element

The naming element is analogous to an assignment statement in other computer language. The assignment

takes place via the naming operation (←). The data name or names on the left side of the naming operator are assigned to the value which results when the expression (Section IV.C.5.d.) on the right of the naming operator is evaluated.

As shown in Syntax Diagram 25 and Example IV - 32, multiple data names can be associated with the same value in on naming element. In the last line of the example the value 0 is assigned to each data name A, B, and C. Example IV - 32 also shows the value of the arithmetic expression B + C assigned to the data name A, and the result of the SIN process being assigned to the data name Y. The third line in Example IV - 32 shows the value which is associated with data name Y being assigned to the A-th element of the array BUF. If an array name is used on the left of the naming operator without specifying a specific element, all elements of the array will be assigned the value of the expression on the right of the naming operator.

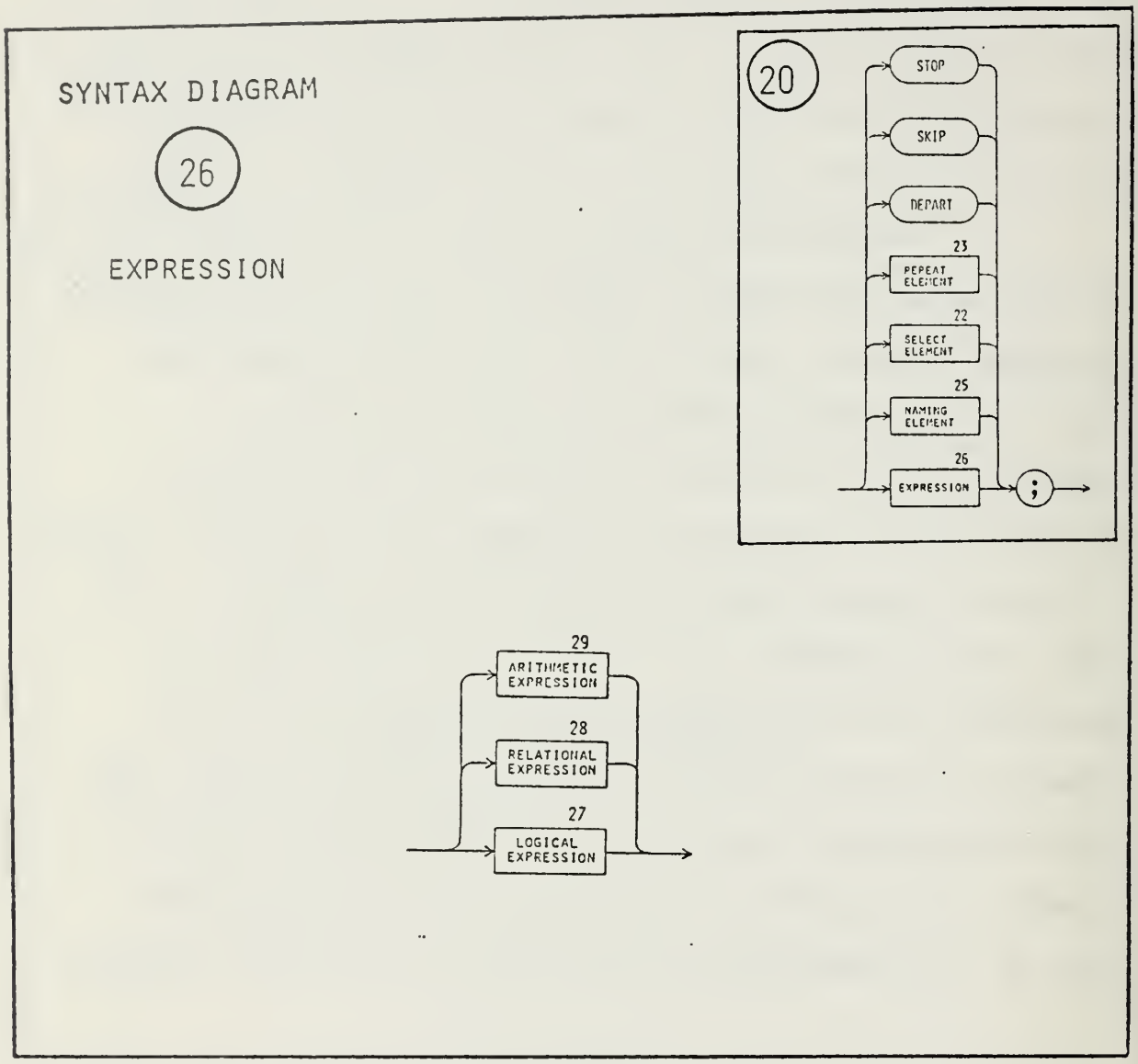


Figure 4-45

d. Expressions

The base language has three types of expressions, the logical expression, the relational expression and the arithmetic expression. The syntactic structure, semantics and use of each are described in the following three sections.

THIS PAGE INTENTIONALLY LEFT BLANK

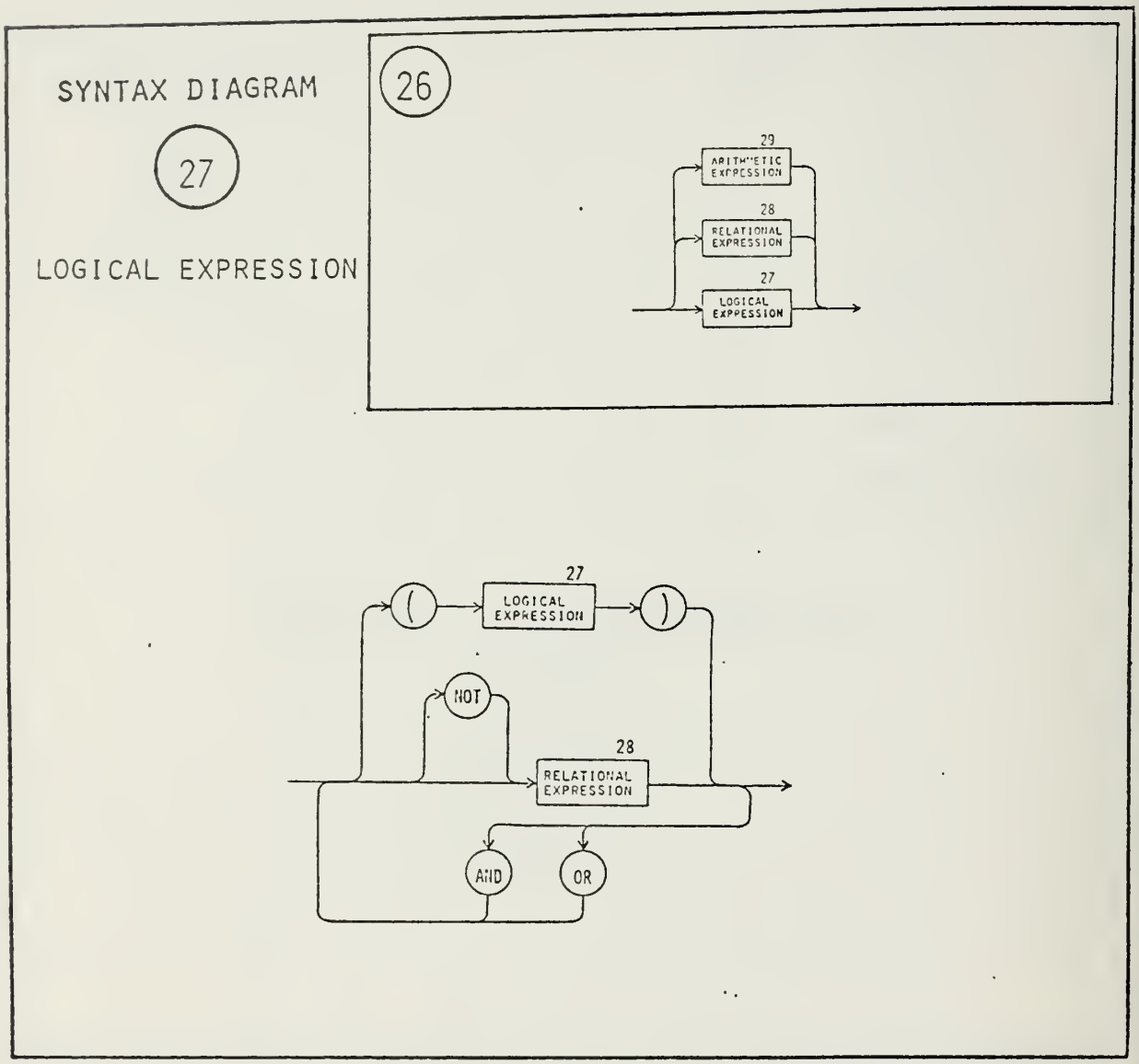


Figure 4-46

Example IV - 33: Sample Logical Expressions

```
A AND NOT B
(X>Y) OR C
```

(1) Logical Expression. A logical expression consists of a sequence of relational expressions separated by the logical operators AND, OR or NOT. It also may consist of only a relational expression, described in the next section

(Section IV.C.5.d.(2)). The NOT logical operator negates the result of the relational expression which it precedes. Complex expressions can be defined with the use of parenthesis. Logical expressions are evaluated from left to right. Parenthesized expressions are completely evaluated prior to the evaluation of the remainder of the expression.

Logical expressions are used in logical labels. Some sample logical expressions are presented in Example IV - 33. The data names A, B, and C used in the samples are assumed to have a boolean property, that is, they can take on values of yes or no, true or false, 1 or 0, etc. In line one, if A is true and B is false the expression will evaluate to true, otherwise it will evaluate to false. In the second line, the relational expression (X>Y) is evaluated first, then the result is logically OR-ed with the calue of C. Both the expression (X>Y) and C must be false for the logical expression to evaluate to false. Otherwise, the logical expression is true.

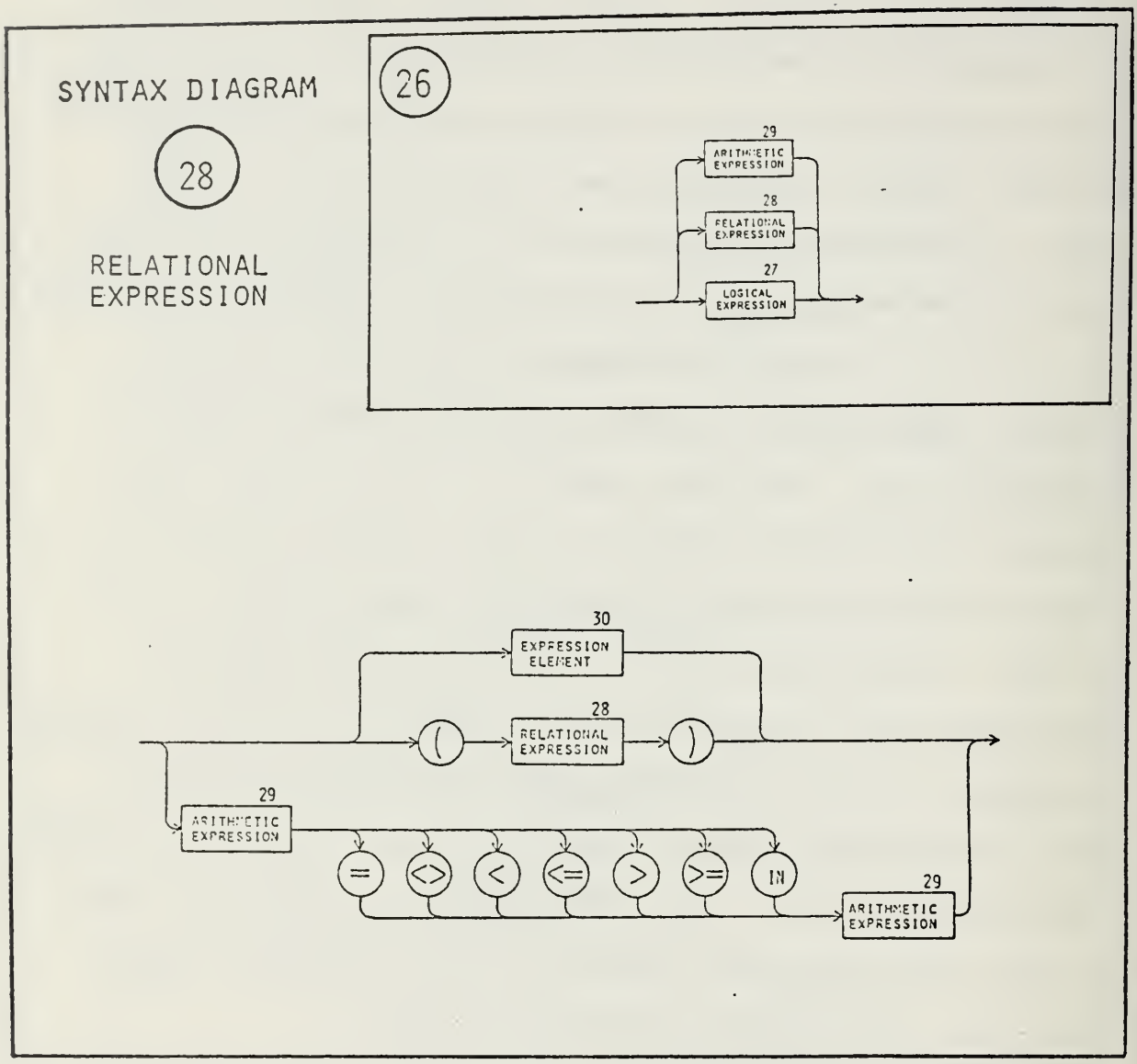


Figure 4-47

Example IV - 34: Sample Relational Expressions

```
A
(C > = D)
X + Y<>J * K
RED IN FLAGCOLOR
```

(2) Relational Expression. The structure of the relational expression is shown in Syntax Diagram 28. A relational expression may be a single expression element

which is described in Section IV.C.5.d.(4) or it may be optionally enclosed in parenthesis. It may also consist of two arithmetic expressions (Section IV.C.5.d.(3)) separated by one relational operator. The meanings of the relational operators are as follows:

= equal	> greater than
<> not equal	>= greater than or equal
< less than	IN contained in (set operator)
<= less than or equal	

Relational expressions are evaluated left to right. Parenthesis can be used to force the complete evaluation of a section of a complex expression prior to evaluation of the remainder of the expression. The primary use of relational expressions is in logical expressions and logical labels.

In Example IV - 34 line one, the data name A is assumed to have a boolean property and represents a relational expression which consists of a single expression element. The second and third lines represent the second and third syntactic structures discussed. The last line represents a relational expression which uses the set relational operator IN. The data name RED is assumed to be a set member and the data name FLAGCOLOR is assumed to be a set. If RED is contained in the set FLAGCOLOR then the expression evaluates to true, otherwise, the result is false.

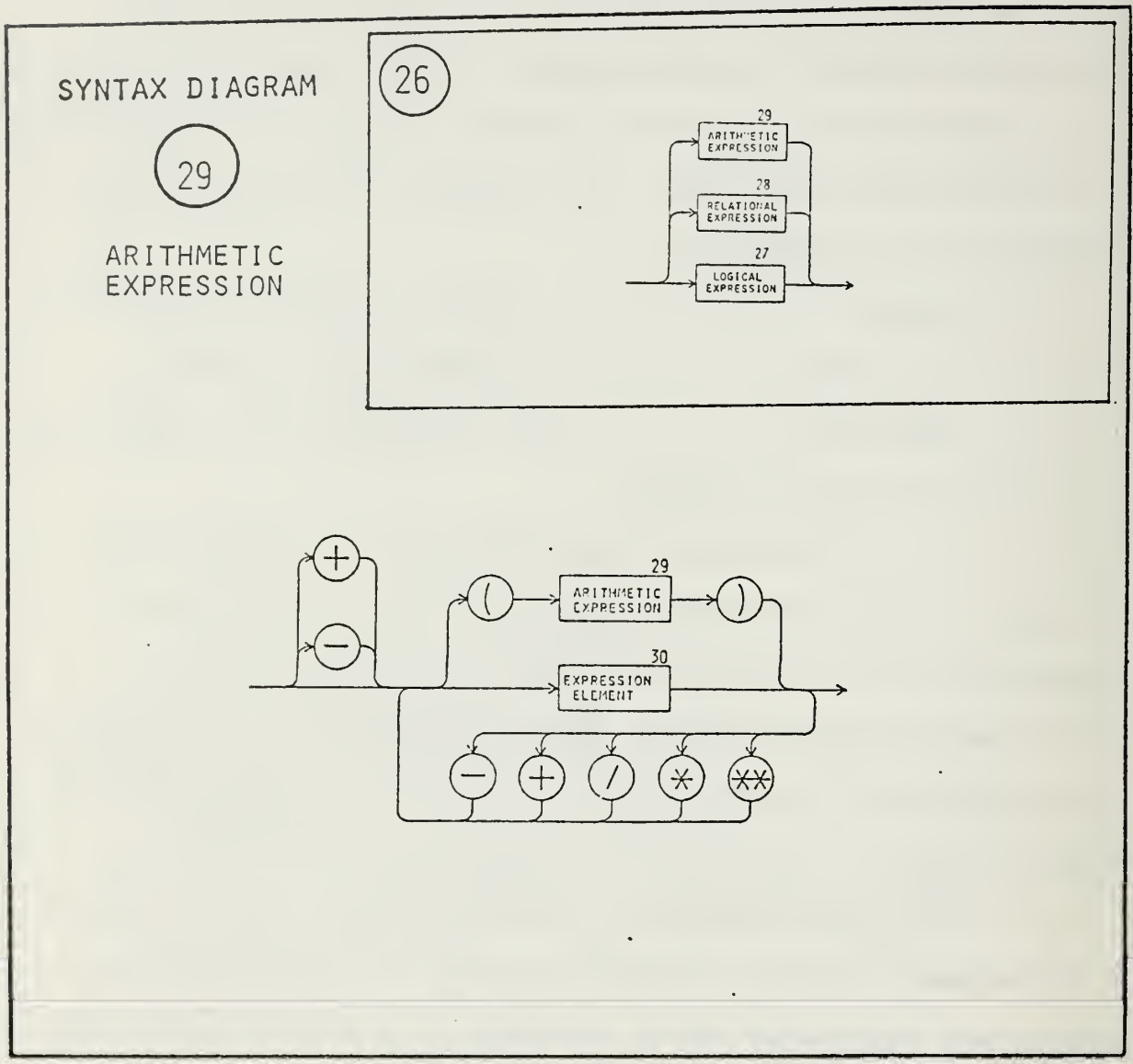


Figure 4-48

Example IV - 35: Sample Arithmetic Expressions

(A + B) * C

(3) Arithmetic Expression. The arithmetic expression is a sequence of expression elements (Section IV.C.5.d.(4)) or arithmetic expressions enclosed in parenthesis separated by infix arithmetic operator symbols. The

basic meaning and precedence of the arithmetic symbols are as follows:

	** exponentiation		
*	multiplication	/	division
+	addition	-	subtraction

The properties of the data names which are operated on by these symbols are user defined and are not part of the base language (see Section IV.C.4.d.), for example, integer, real, double precision, etc. Because of this, the precise programmatic function of these operators cannot be built into the base language. User defined or basic processes such as integer arithmetic processes or real arithmetic processes may be declared and linked to arithmetic symbols through the LINK directive. A complete discussion of the purpose and use of the LINK directive is contained in Section IV.C.6.c.

Example IV - 35 shows an arithmetic expression. The precise function of the + and * operators is dependent on the properties of the data names and the definition of the linked arithmetic processes.

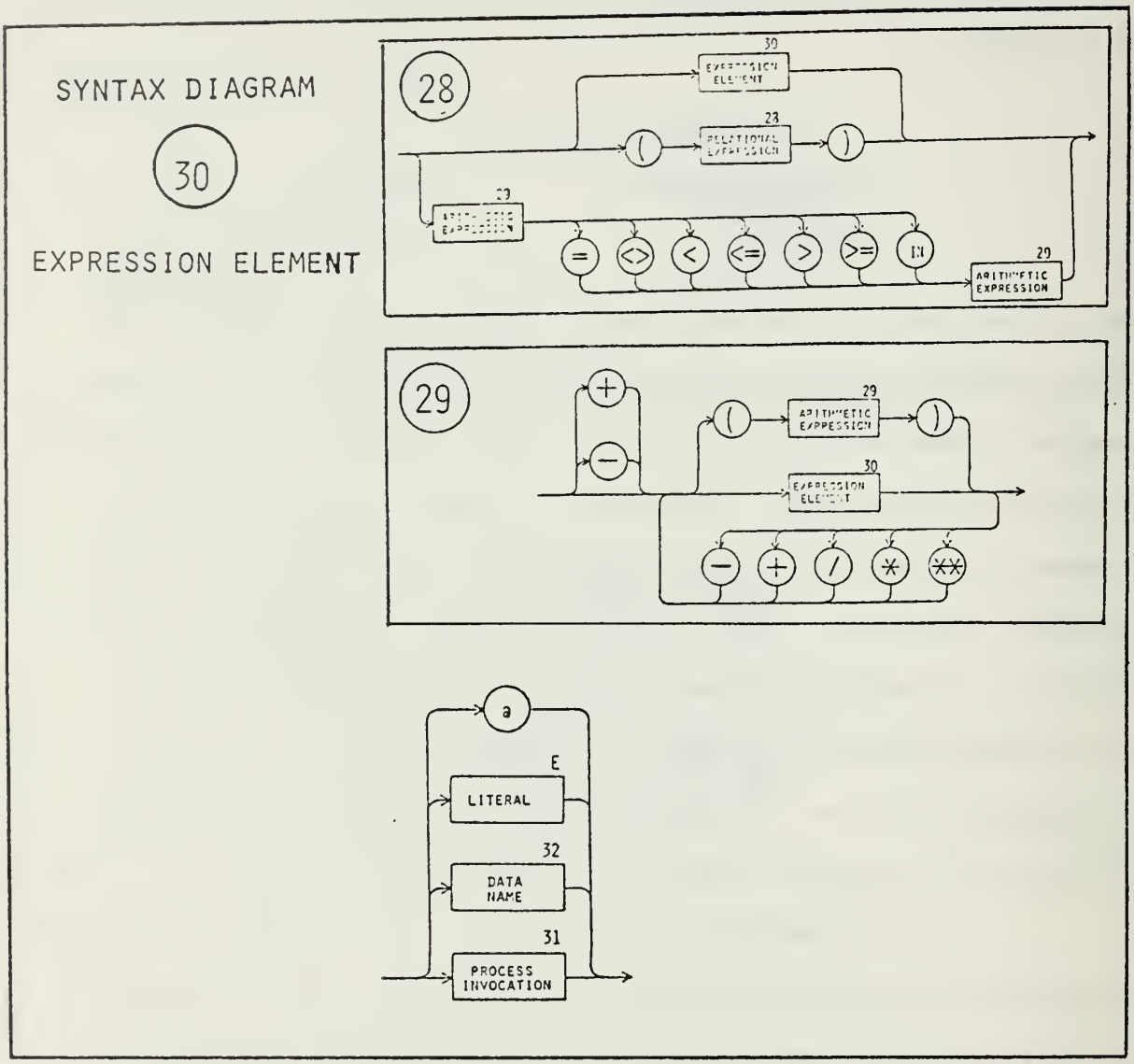


Figure 4-49

(4) Expression Element. The expression element is the basic element of a relational or arithmetic expression. The expression element syntax category is composed of four elements: the process invocation, the data name, the literal and the answer represented by the symbol @. These elements are discussed in the following four sections.

THIS PAGE INTENTIONALLY LEFT BLANK

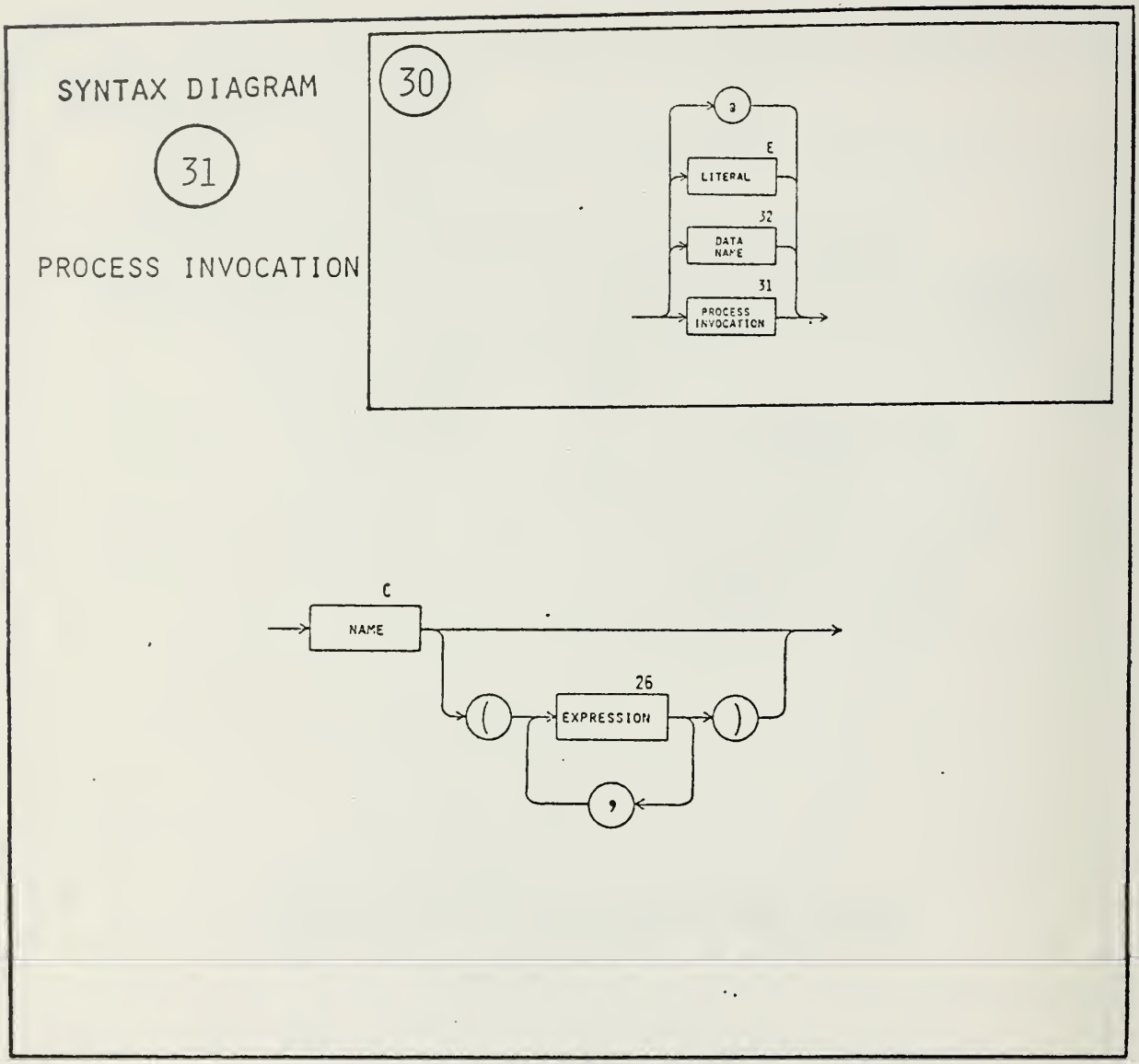


Figure 4-50

Example IV - 36: Sample Process Invocations

```
SAMPROC (A, ., C - D);
ANAME←APROC (X, Y, Z);
```

(a) Process Invocation. Process invocation is a calling of the named process for execution. It is analogous to a subroutine call or precedure call in other computer languages such as FORTRAN or PL/I. The process

invocation causes execution to transfer from the point of the invocation to the beginning of the named process. Upon completion of execution of the invoked process control is transferred back to the invoking process. Execution resumes in the invoking process at the next executable instruction after the process invocation.

As shown in Syntax Diagram 31, a parameter list may optionally follow the process name. The parameter list is composed of a sequence of expressions. The values associated with the elements in the parameter list are passed to the invoked process via the input interface mechanism. The number and sequence of the elements in the parameter list must match the number and sequence of elements in the input interface of the invoked process. If there are not as many parameter values passed into a process as the input interface is defined to accept, or some input parameters are irrelevant to a particular use, a place holder may be used in the parameter list. This is shown in the first line of Example IV - 36. The place holder is a single period symbol (.) and is defined in Section IV.C.5.d.(4)(b). The values associated with the invoked process output interface names are returned to the invoking process via the output interface mechanism. These values may be used or ignored by the invoking process. Output parameters are used by assigning them a name in the invoking process via the naming element (Section IV.C.5.c.) as shown in the second line of Example IV - 36. The output

values may also be directly accessed immediately after return from the invoked process through the use of the answer element (Section IV.C.5.d.(4)(d)). The output values may be ignored by not using them as described above. The first line of Example IV - 36 shows an example of this, assuming that the SAMPROC process contains an output interface section.

THIS PAGE INTENTIONALLY LEFT BLANK

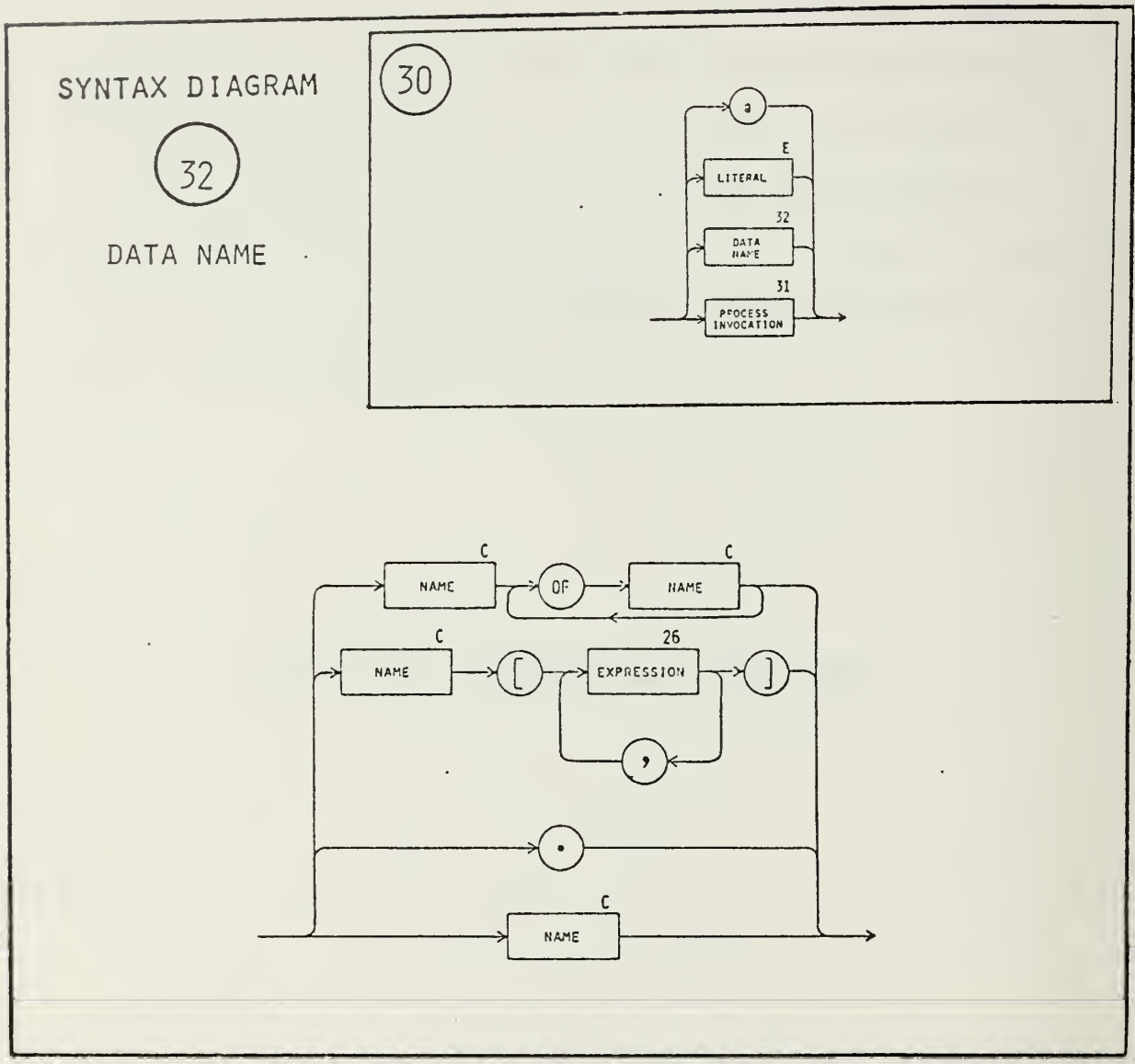


Figure 4-51

Example IV - 37: Sample Data Names

```
ARF
BUFFER [N]
NAME OF FORM88
A, ., C←ARFETT (X,Y,.,Z);
```

(b) Data Name. A data name is the name of an expression element. It may be the name of a data item, a component of a data item such as an array element, or

composite element, or it may be a process name. A data name may also be a period symbol (.) which is used as a place holder for aligning name lists to the input and output interface sections of an invoked process. A data name is associated with a specific data value or process and provides access to either of these.

Example IV - 37 shows, in order, some simple examples of a name, an array element reference and a composite element reference. The name ARF may refer to a data item or a process. The data name BUFFER [N] refers to the N-th element of the single dimension array named BUFFER. The data name NAME OF FORM88 refers to the NAME field of the composite data item FORM88 which is defined in Example IV - 19 in Section IV.C.4.d.(2). The last line in Example IV - 37 shows the use of the place holder symbol to align the input parameters, X, Y, and Z to the first, second and fourth input interface elements of the ARFETT process. It also shows the alignment of the output data names A and C to the first and third elements in the output interface of the ARFETT process.

(c) Literal. Literals are tokens which represent constant values. They consist of literal strings of characters and signed or unsigned numbers. The syntactic structure, semantics, and use of literals is discussed in Section IV.C.7.c.(3).

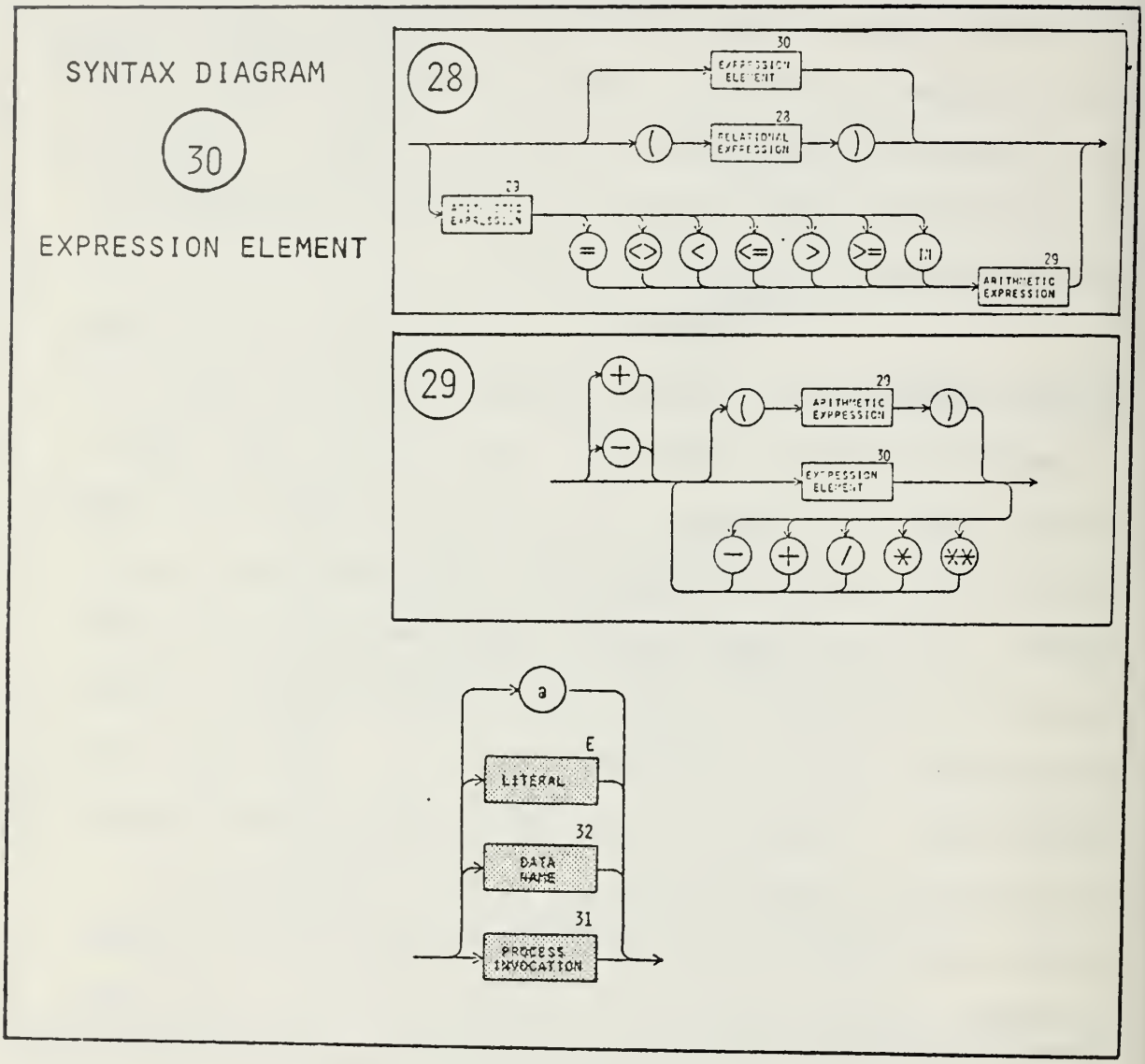


Figure 4-52

(d) Answer. An answer is the immediate result of an expression or the output of a process. The answer is represented by the symbol @. The special name @ is associated with the result of an expression or results of a process only until the next expression is executed or process is invoked.

The answer element is primarily a tool for interactive process execution in an interpretive environment. However, it can be used effectively in a translation environment as shown in Example IV - 29, the select element example in Section IV.C.5.b.(2). In this case the answer symbol represents the result of the expression (A + B) for each of the select logical labels.

SYNTAX DIAGRAM

33

DIRECTIVE

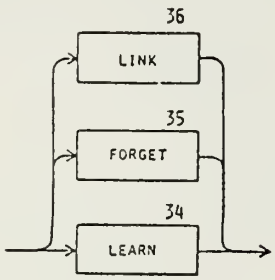


Figure 4-53

6. Directive

A directive is a direction to the base language translator to perform a specific action. The directives which are available in the base language are: the learn directive which is described in Section IV.C.6.a., the forget directive which is described in Section IV.C.6.b., and the link directive which is described in Section IV.C.6.c

The learn and forget directives cause the translator to interact with the system library. Respectively, they cause designated processes or data descriptions to be stored in the system library for long-term future reference or erased from the library. The link directive associates a user defined arithmetic process with an infix arithmetic operation symbol.

During translation if a referenced process is not defined in the translation group, or if a referenced data property is not imported or defined in the associated process, then the library is searched for the appropriate name. If the item is found, it is made a part of the translation group or process. If the item is not found, then a translation error is generated.

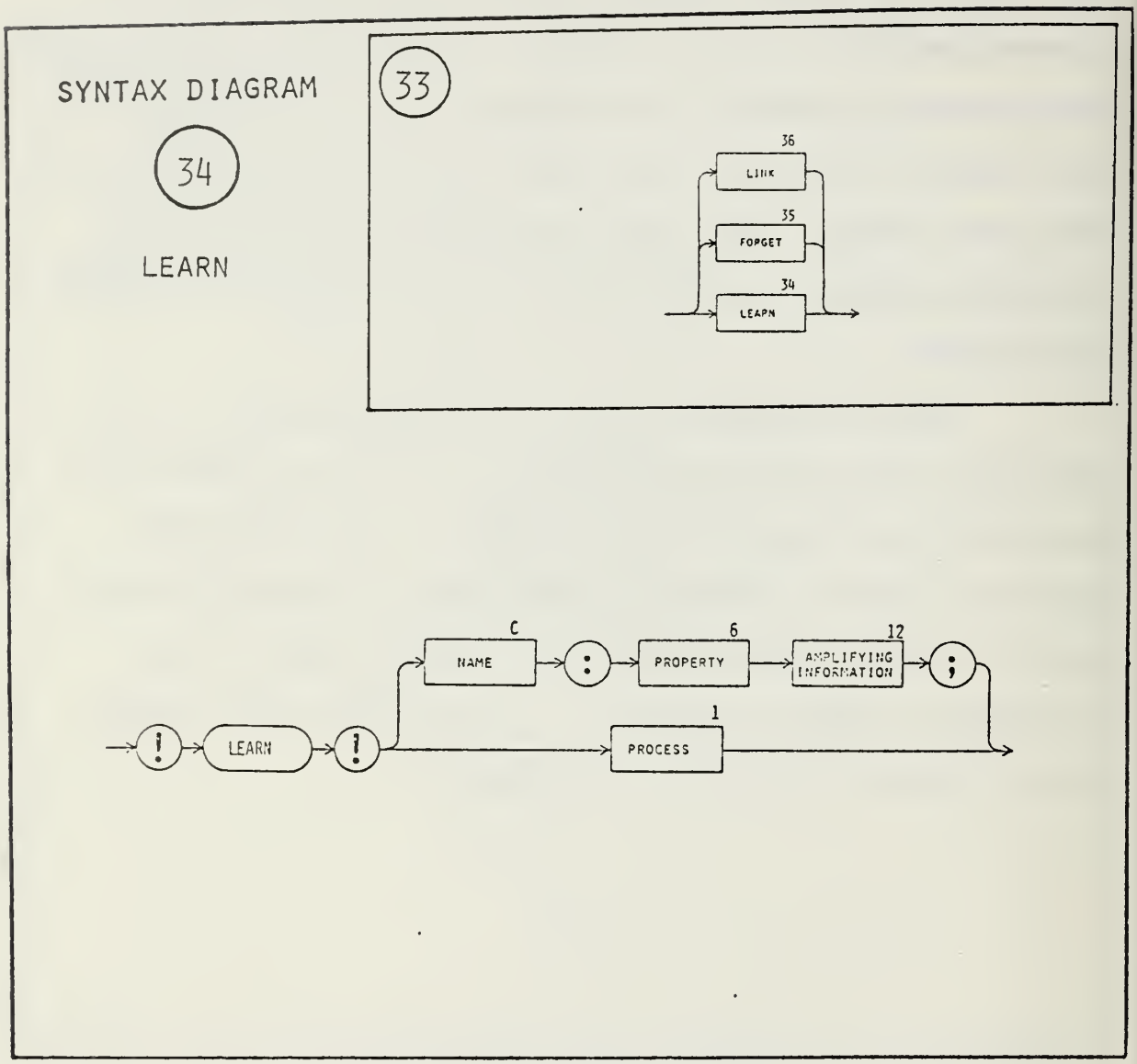


Figure 4-54

Example IV - 38: Sample Learn Directives

```
!LEARN! EXCHANGE PROCESS
        INPUT A, B END INPUT
        OUTPUT B, A END OUTPUT
        END PROCESS

!LEARN! INTEGER: ARRAY [16] BITS, LIMIT (0..65535);
```

a. Learn Directive

The LEARN directive causes a specified process or data declaration to be stored into the system library. It is used to create an easily accessible collection of commonly used processes or data structures. The syntactic structure of the LEARN directive is shown in Syntax Diagram 34 and Example IV - 38. The construct used to learn a data structure is very similar to a data description with the restriction that only one name may be associated with the property and amplifying information. The LEARN directive may be used in the data definition section of a process or as an individual element in a translation group. To preserve the independence of processes which are saved in the system library, the LEARN directive may not be used in conjunction with a process which uses the IMPORTED property within its data definition section.

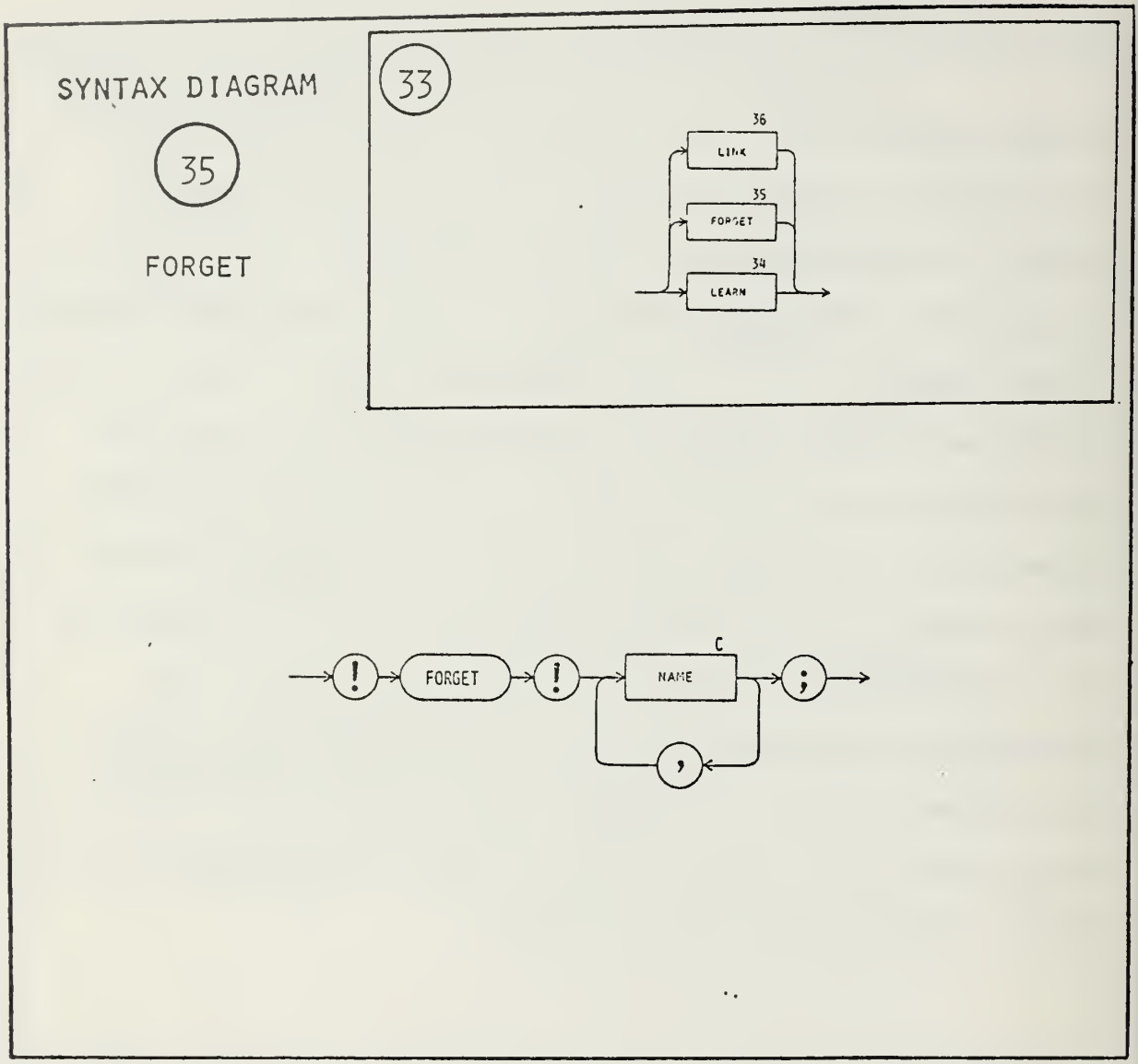


Figure 4-55

Example IV - 39: Sample Forget Directives

```
!FORGET! EXCHANGE;
!FORGET! INTEGER;
```

b. Forget Directive

The FORGET directive causes the specified process or data description to be erased from the library. Example IV - 39 shows the use of the FORGET directive to

erase the EXCHANGE process and INTEGER data description which were defined in Example IV - 38. The FORGET directive is used as an individual element in a translation group.

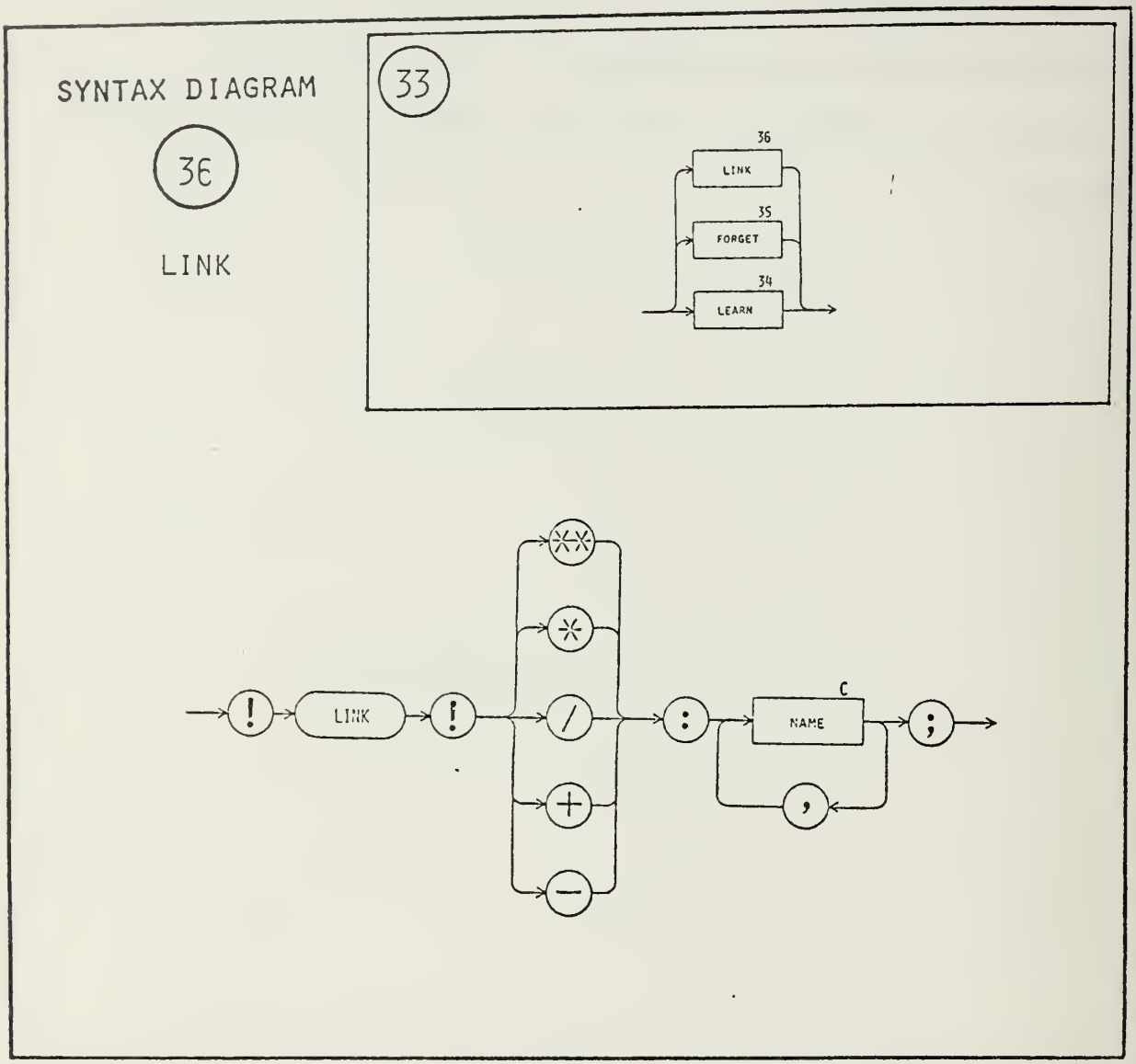


Figure 4-56

Example IV - 40: Sample Link Directive

```
!LINK! * : REALMULT, INTMULT, MIXMULT;
```

c. Link Directive

The LINK directive is used to link a special infix arithmetic operation symbol to a user defined arithmetic process. When a process has been linked to an

arithmetic operator symbol, the symbol may be then used as an infix operator instead of the arithmetic process name. Processes which are linked to an arithmetic symbol are restricted to two input parameters and one output parameter in the process interfaced due to the binary property of arithmetic operators.

During translation, when an arithmetic operator symbol is encountered, the properties of the associated data names are determined and compared to the properties of the input parameters defined in the processes which are linked to the arithmetic operator symbol. Matching properties determine the arithmetic process which is to be associated with the symbol and the order in which the data names are passed to the arithmetic process. For example, assume the expression A * B was encountered during translation and that A has the property INTEGER and B has the property REAL. Assume also that the arithmetic operator * was linked as in Example IV - 40 to processes which perform real number multiplication (REALMULT), integer number multiplication (INTMULT), and mixed real and integer multiplication (MIXMULT). Also, the input interface and data definition sections of these processes are defined as shown in Examples IV - 41, IV - 42 and IV - 43.

Example IV - 41: Sample of a partial real multiplication process:

```
REAL MULT PROCESS
    INPUT X, Y END INPUT
    DATA X, Y: REAL;
        ANS: REAL;
        .
        .
        .
    END DATA
    .
    .
    .
    OUTPUT ANS END OUTPUT
END PROCESS
```

Example IV - 42: Sample of a partial integer multiplication process:

```
INTMULT PROCESS
    INPUT X, Y END INPUT
    DATA X, Y: INTEGER;
        ANS: INTEGER;
        .
        .
        .
    END DATA
    .
    .
    .
    OUTPUT ANS END OUTPUT
END PROCESS
```

Example IV - 43: Sample of a partial mixed real and integer multiplication process:

```
MIXMULT PROCESS
    INPUT X, Y END INPUT
    DATA X: REAL;
        Y: INTEGER;
        ANS: REAL;
        .
        .
        .
    END DATA
    .
    .
    .
    OUTPUT ANS END OUTPUT
END PROCESS
```

When the expression A * B is encountered during translation the properties of A and B are determined to be INTEGER and REAL respectively. The processes which are linked to the symbol * are then individually examined to determine if the properties of the input parameters match the properties of A and B. In this example the properties of the input parameters in the MIXMULT process match but are in the reverse order. This indicates to the translator that A and B must be exchanged prior to their use in the MIXMULT process. The exchange can be done through the use of the EXCHANGE process (Example IV - 38, Section IV.C.6.a.) in conjunction with the MIXMULT process. The * symbol is now associated with the EXCHANGE and MIXMULT process for the expression A * B.

7. Lexical Structures

a. Character Set

The process text consists of a sequence of characters taken from an implementation oriented character set. The base language assumes the character set includes, at least, the digits 0 through 9, the upper case letters A through Z, the space character and the following symbols:

! ( ) [ ] { } ← * / + - = < > . , ; : ' # @ _

This character set is implementation oriented and may require modification during implementation of the base language. It should be noted that each of the above characters is in the 64 character ASCII subset except { } and ←.

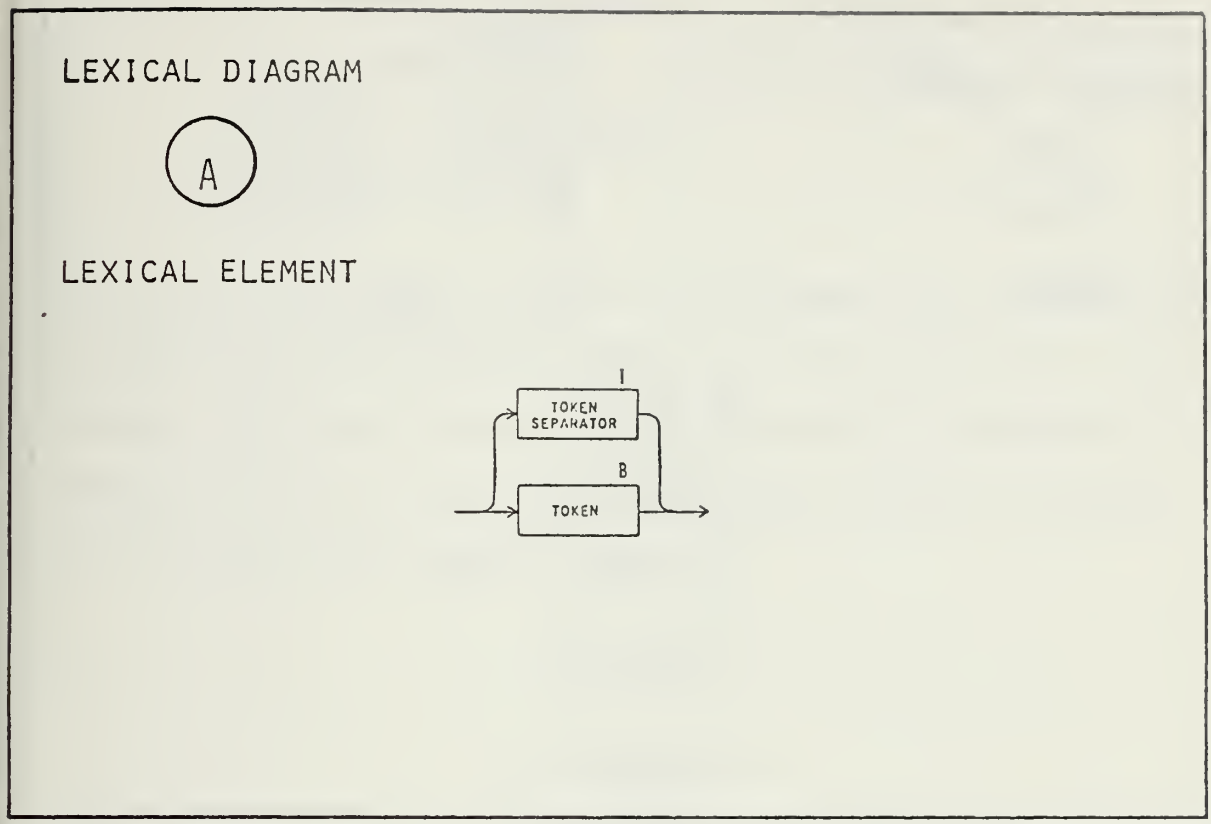


Figure 4-57

b. Lexical Elements

A lexical element is a sequence of characters which make up the text of a process. Lexical elements are separated into two categories, tokens and token separators. The syntactic and semantic correctness of a process is based on the sequence of lexical elements which it contains. Token separators as the name implies act as separators between contiguous tokens. These two categories are described in detail in the following sections.

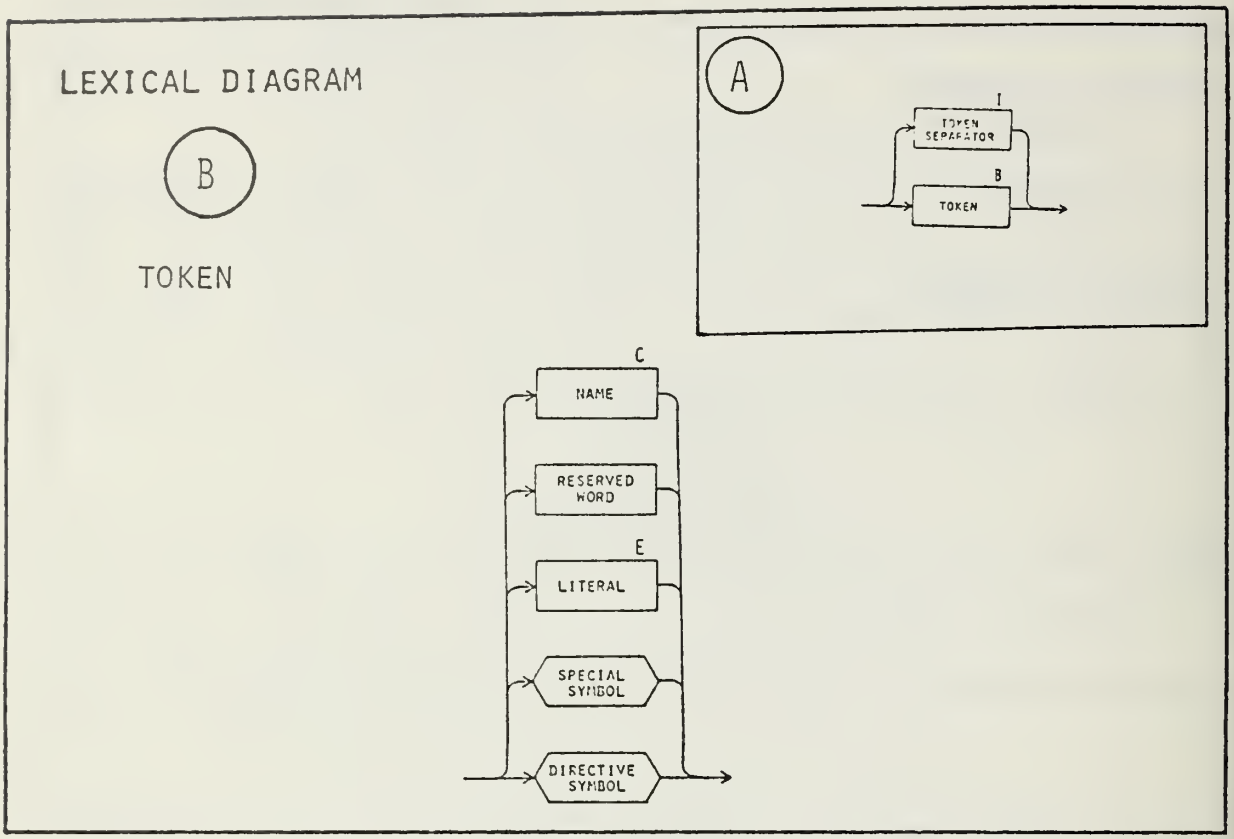


Figure 4-58

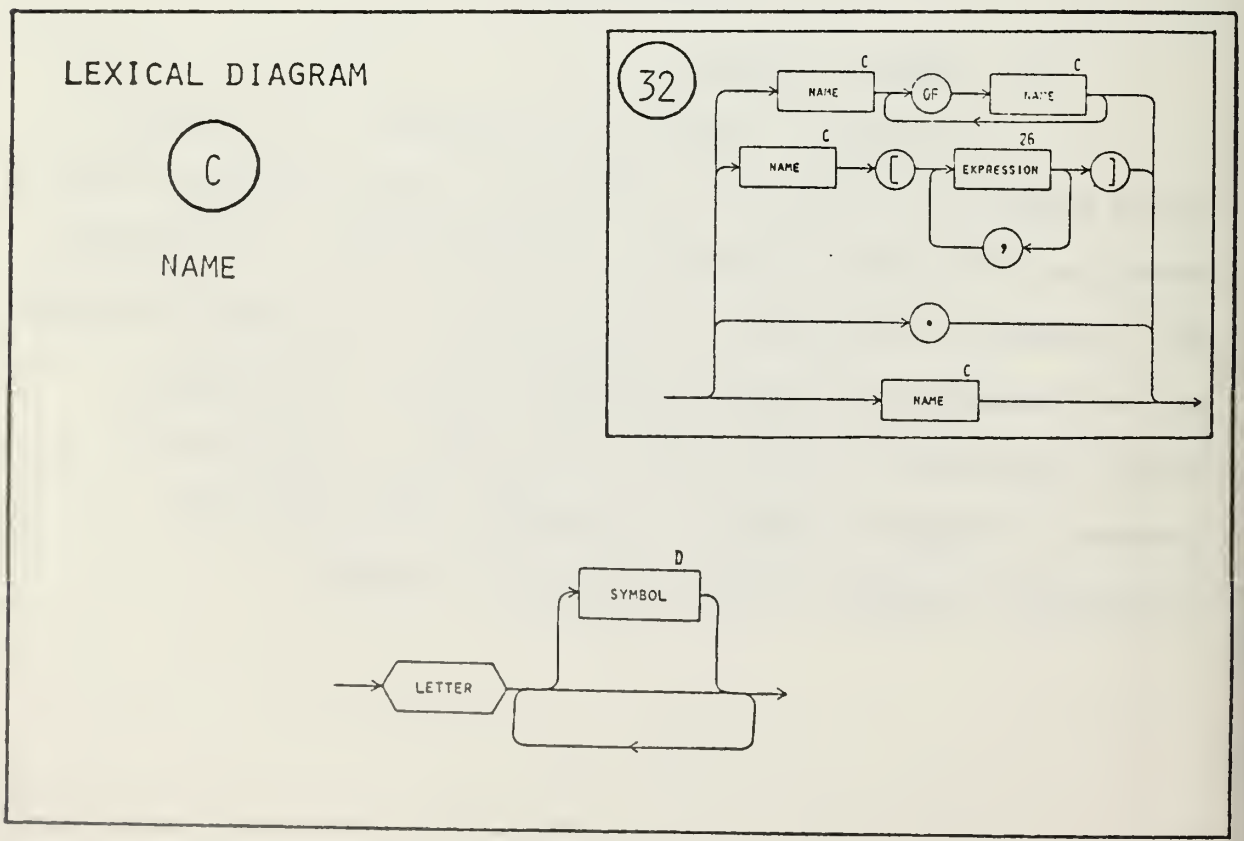


Figure 4-59

c. Tokens

Example IV - 41: Sample Names

```
SAMPLE NAME
NAME23
A123XYZ
```

(1) Name. A name is analogous to an identifier in many other computer languages. A name must start with a letter. The first letter may be followed by a sequence of symbols which are described in the next section. Example IV - 41 shows some sample names.

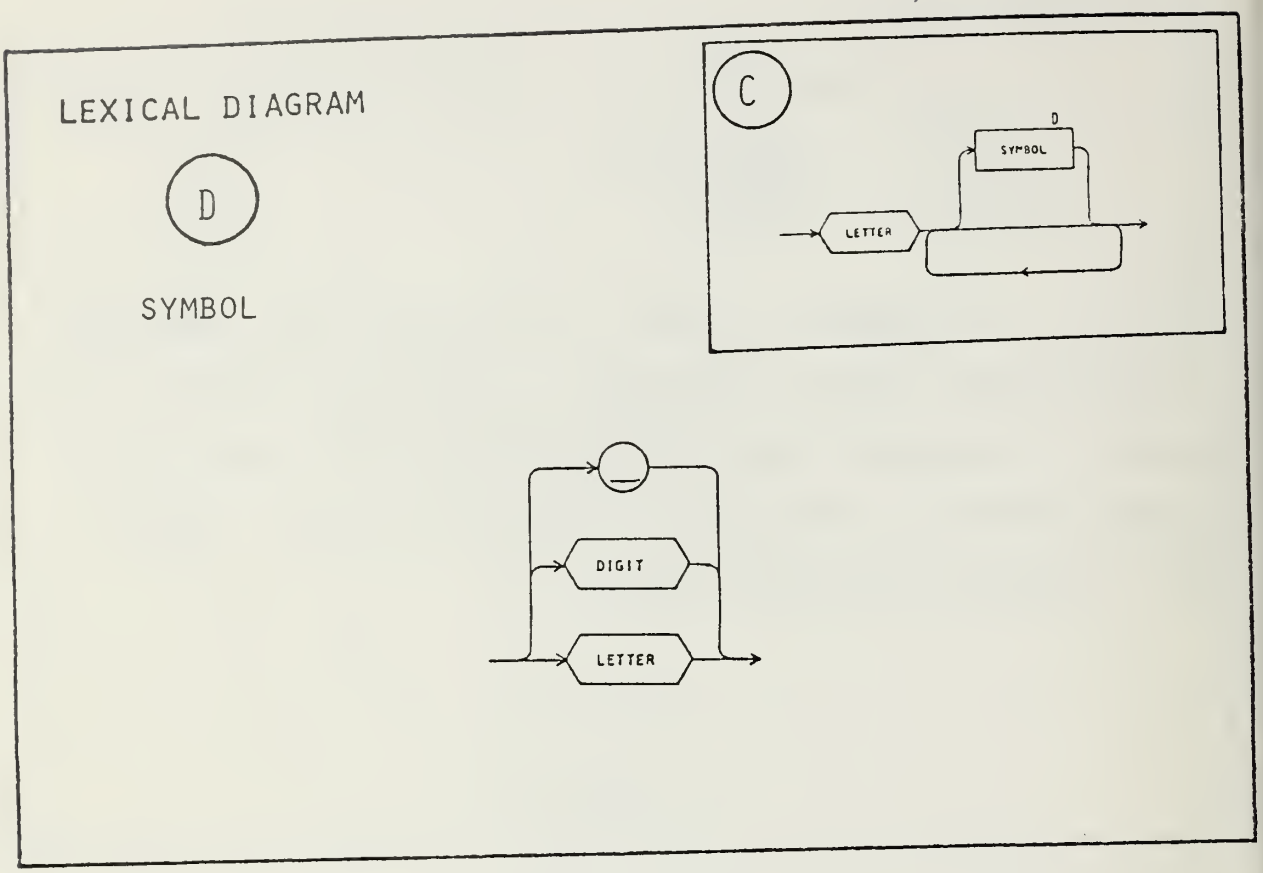


Figure 4-60

A symbol may be a letter, digit or underscore symbol. Valid letters consist of the characters A through Z. Valid digits consist of the decimal digits 0 through 9. The underscore symbol (_) is used as a break symbol in user defined names.

(2) Reserved Words. A reserved word is a token which has the form of a name. Because of its syntactic role in the base language, a reserved word can be used in a process only in the context established by the syntax diagrams in which they appear. Table 4-1 lists the reserved words of the base language.

ARRAY	BYTES	IMPORTED	LIST	SELECT	DATA
AND	COMPOSITE	INITIAL	NOT	SET	DEFAULT
BIT	CONSTANT	INPUT	OR	SKIP	DEPART
BITS	CONTINUE	LEARN	OUTPUT	STOP	OPERATION
BY	END	LINK	PROCESS	TERMINATE	VALUE
BYTE	FORGET	LIMIT	REPEAT	TIMES	WITH

TABLE 4-1. Base Language Reserved Words

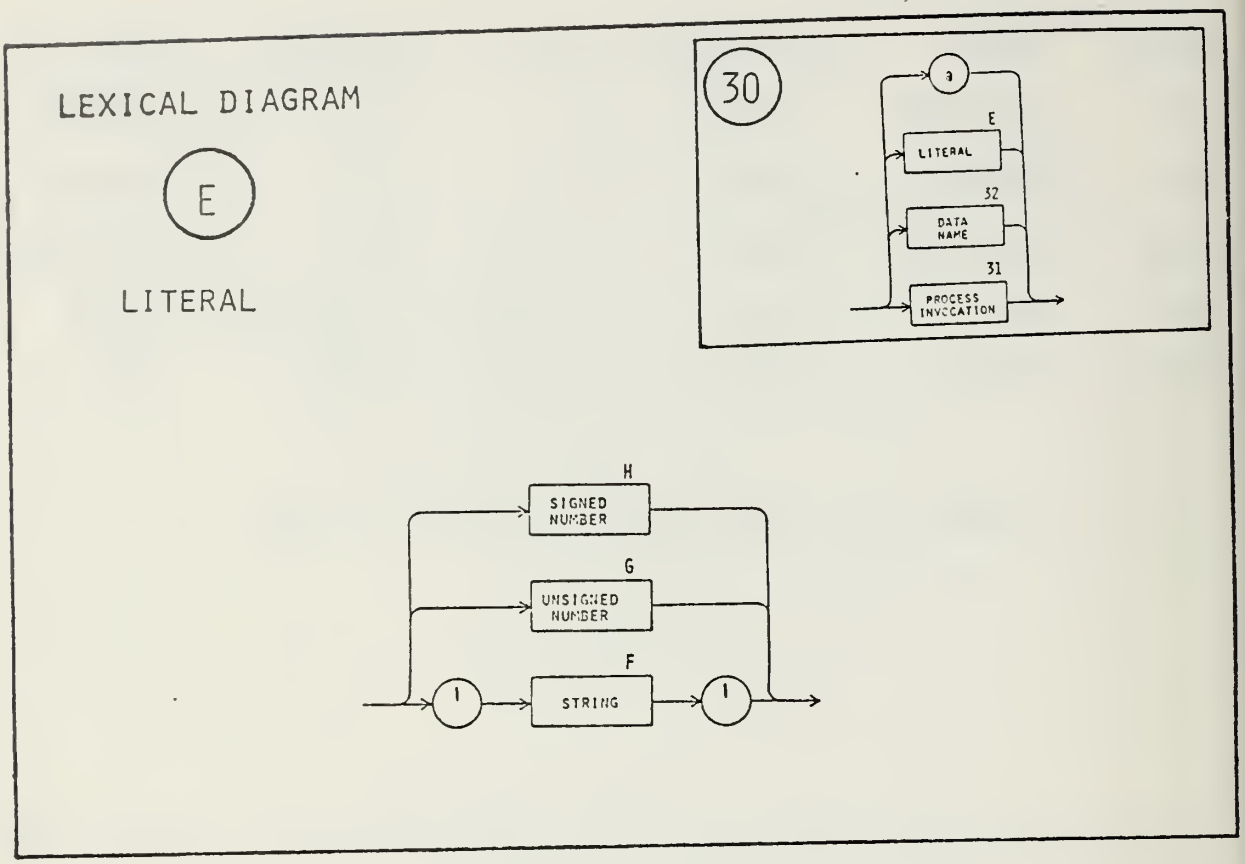


Figure 4-61

Example IV - 42: Sample Literals

```
'THIS IS A LITERAL STRING'
    27.542
    -11.3 E-12
```

(3) Literal. Literals are tokens which represent constant values. A literal may be a string (Section IV.C.7.c(3)(a)) enclosed in single quotes, an unsigned number (Section IV.C.7.c.(3)(b)), or a signed number (Section IV.C.7.c.(3)(c)). A literal string is stored internally as a sequence of symbol codes, for example ASCII codes. There is no change in representation of the symbols in a string between the internal and external representation.

Signed and unsigned numbers are stored internally as constant values. There is a conversion between external symbol codes and internal machine representation by the base language translator. Example IV - 42 provides some sample literals. The first line represents literal string. The other two lines represent unsigned and signed numbers respectively.

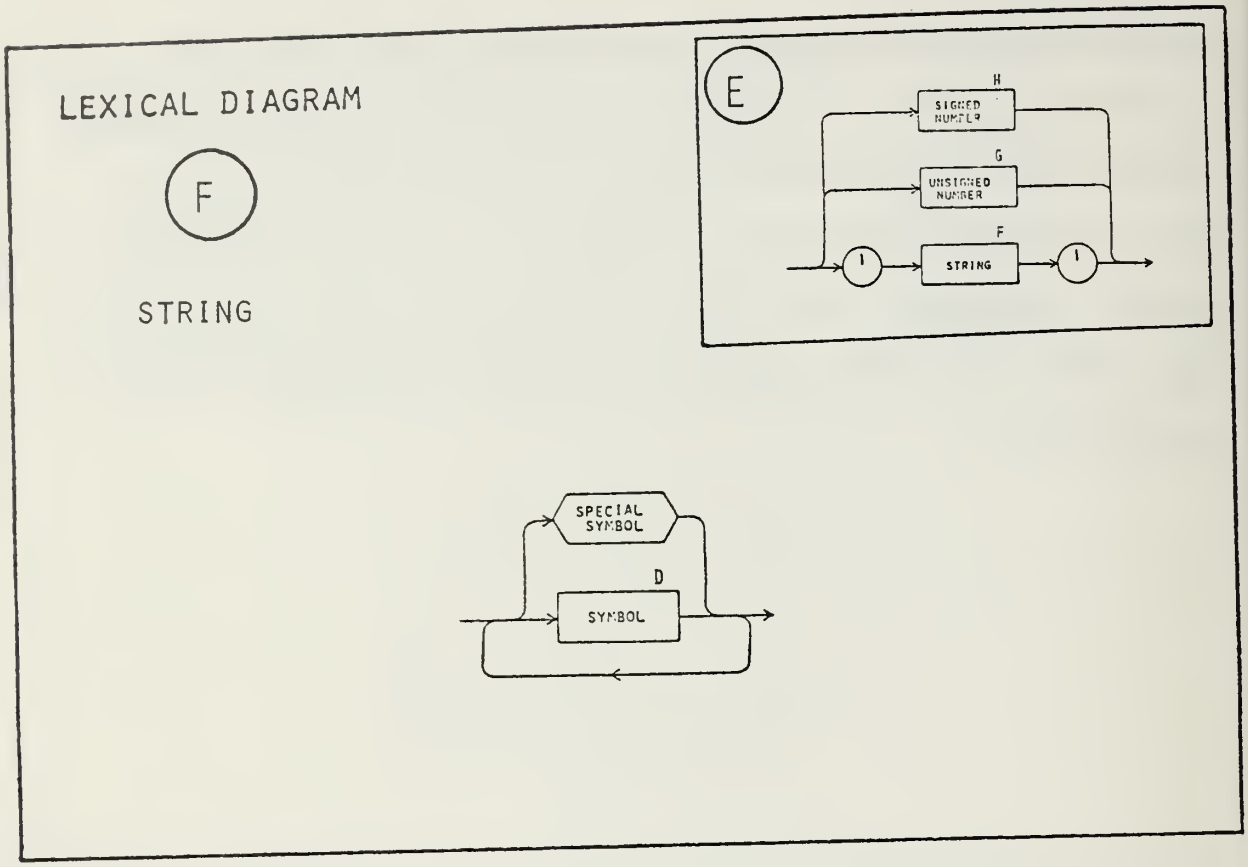


Figure 4-62

Example IV - 43:

'THIS IS A LITERAL STRING'

(a) String. A string is a sequence of one or more symbols (Section IV.C.7.c.(1)) or special symbols. Special symbols are described in Section IV.C.7.c.(4).

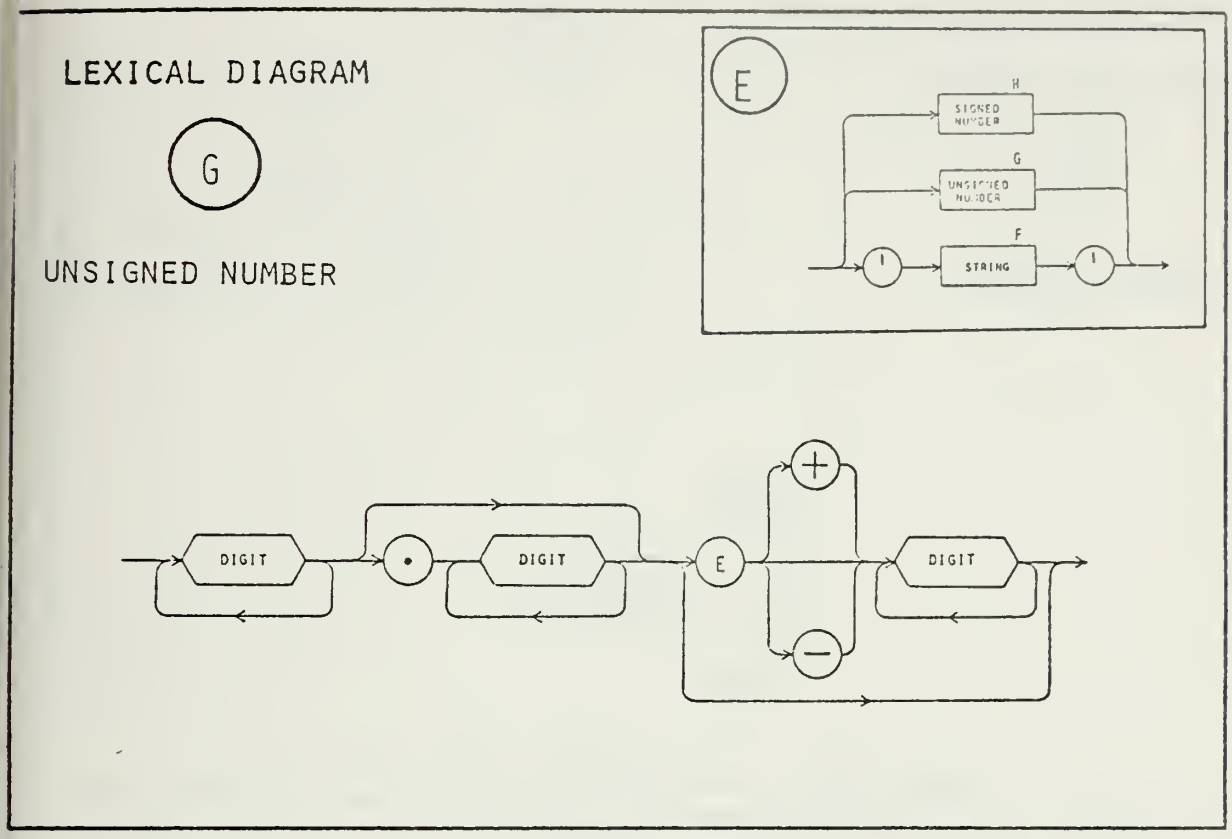


Figure 4-63

Example IV - 44: Sample Unsigned Numbers

```
10
5.32
27.5 E 32
```

(b) Unsigned Number. An unsigned number is a sequence of digits optionally followed by a period symbol (.) and a sequence of one or more digits. Scientific notation, base 10 is optional and is denoted by the character E followed by an optional + or - symbol and a sequence of digits. If neither a + or - symbol is used in the scientific notation representation the default is positive (i.e., + is assumed). Example IV - 44 provides some sample unsigned numbers.

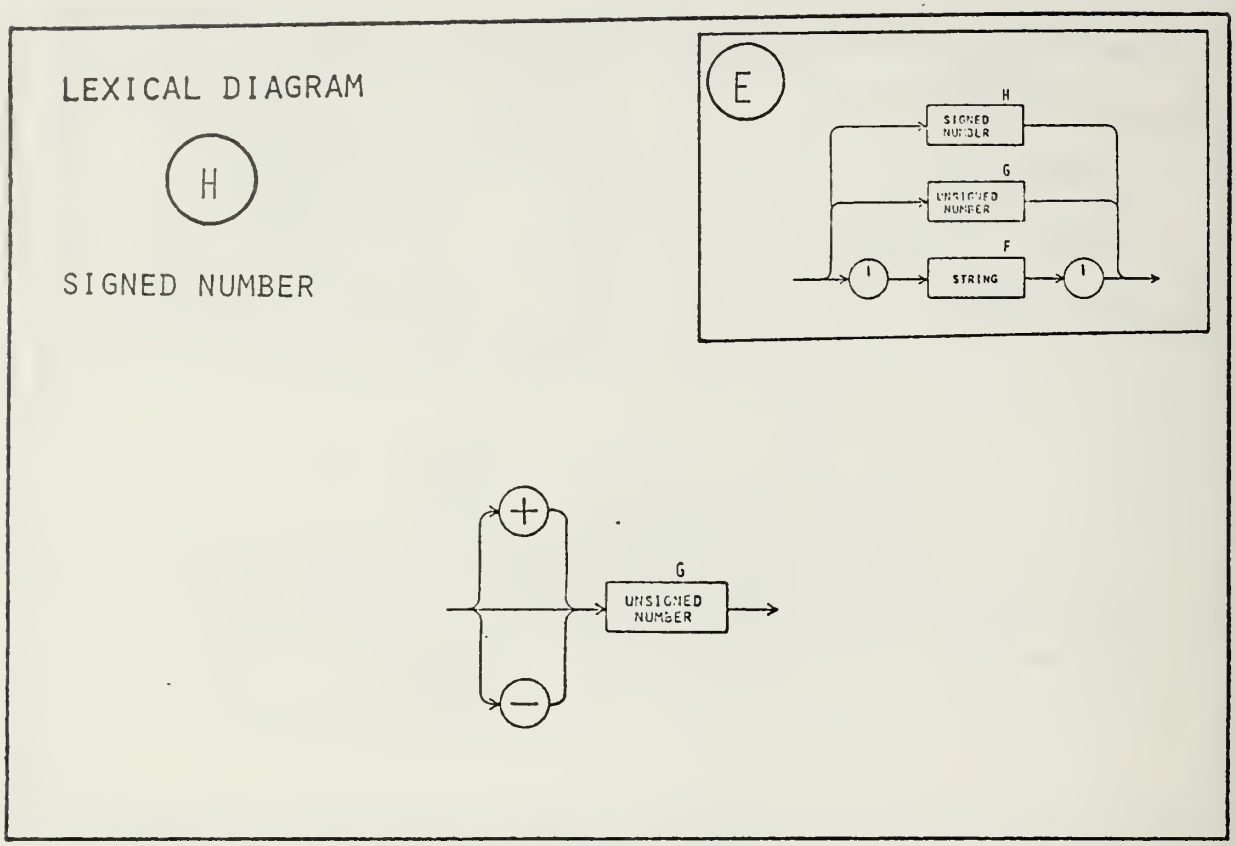


Figure 4-64

Example IV - 45: Sample Signed Number

```
+10
-15.3 - 4
 23.1
```

(c) Signed Number. A signed number is an unsigned number preceded by an optional + or - symbol. If neither a + or - symbol is used than + is assumed.

THIS PAGE INTENTIONALLY LEFT BLANK

(4) Special Symbols. Table 4-2 lists the special symbols used in the base language and their uses.

SPECIAL SYMBOL	USE
! !	TRANSLATOR DIRECTIVE
( )	PARENTHESIZATION
[ ]	ARRAY DEFINITION AND REFERENCE
{ }	OPERATION ELEMENT GROUP
←	NAMING OPERATOR
. .	BOUNDS SEPARATOR
.	DATA NAME PLACE HOLDER
,	GENERAL SEPARATOR IN LISTS
;	TERMINATION SYMBOL FOR DATA DECLARATION AND OPERATION ELEMENT
:	SEPARATOR FOR LOGICAL OR RELATIONAL CONDITIONS AND DATA NAME DECLARATION
'	LITERAL STRING DELIMITER
@	ANSWER
#	COMMENT DELIMITER
_	NAME SEPARATOR SYMBOL
**	EXPONENTIATION
*	MULTIPLICATION
/	DIVISION
+	ADDITION
-	SUBTRACTION
=	EQUAL
<>	NOT EQUAL
<	LESS THAN
<=	LESS THAN OR EQUAL
>	GREATER THAN
>=	GREATER THAN OR EQUAL

TABLE 4-2. Special Symbols and Their Uses

(5) Directive Symbol. The exclamation symbol (!) is used to indicate a translator directive. The exclamation symbol surrounds a reserved word which indicates a special action to be taken by the base language translator. Descriptions of directives and examples are contained in Section IV.C.6.

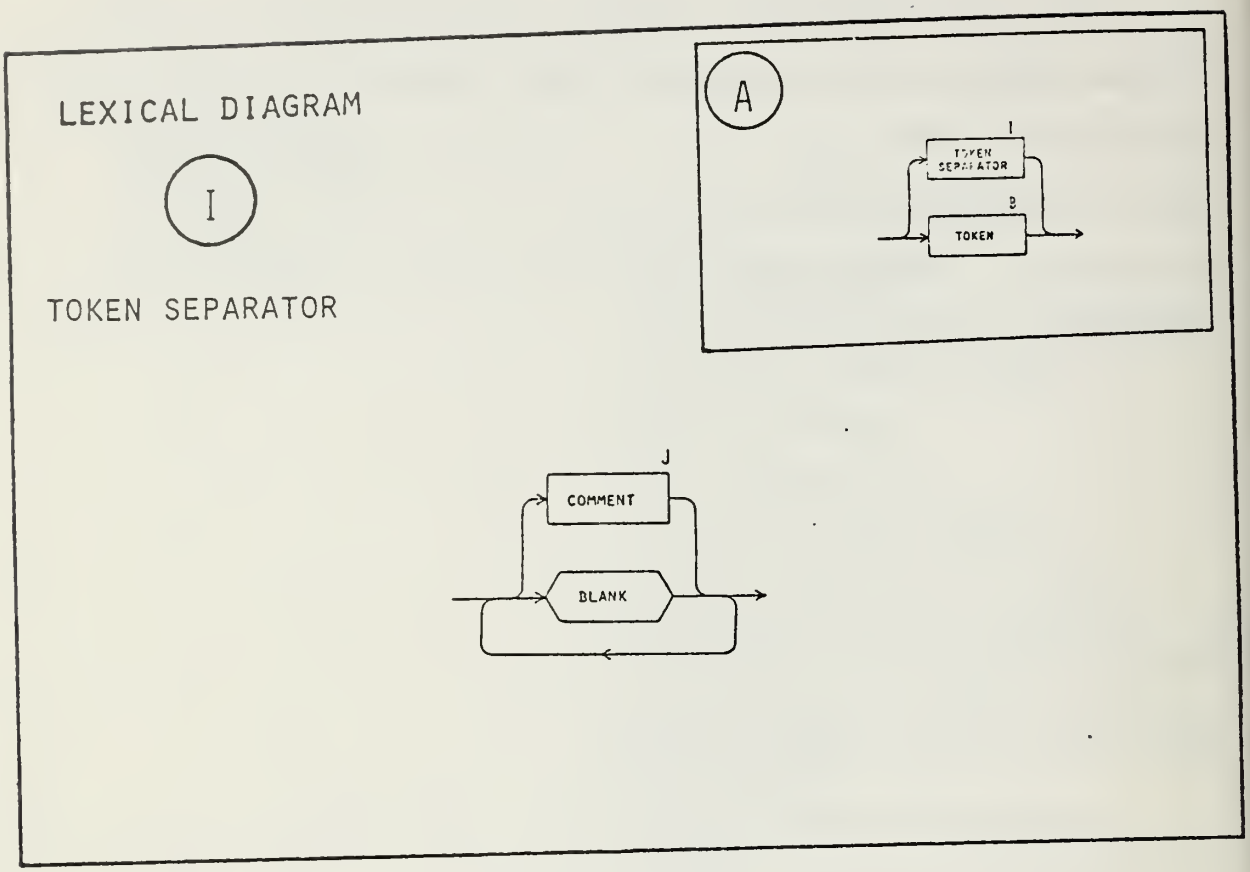


Figure 4-65

Example IV - 46: Sample Exchange Process

```
EXCHANGE PROCESS
INPUT A, B END INPUT
OUTPUT B, A END OUTPUT
END PROCESS
```

d. Token Separator

A token separator consists of a sequence of blank symbols or comments. It may appear between any two tokens. If the juxtaposition of two tokens in the sequence of tokens would produce a sequence of characters which make up a longer token then a token separator must appear between those tokens. As an example, a token separator must appear

between END and PROCESS and may optionally appear between a name and a comma as shown in Example IV - 46.

(1) Blank. A blank is a space character.

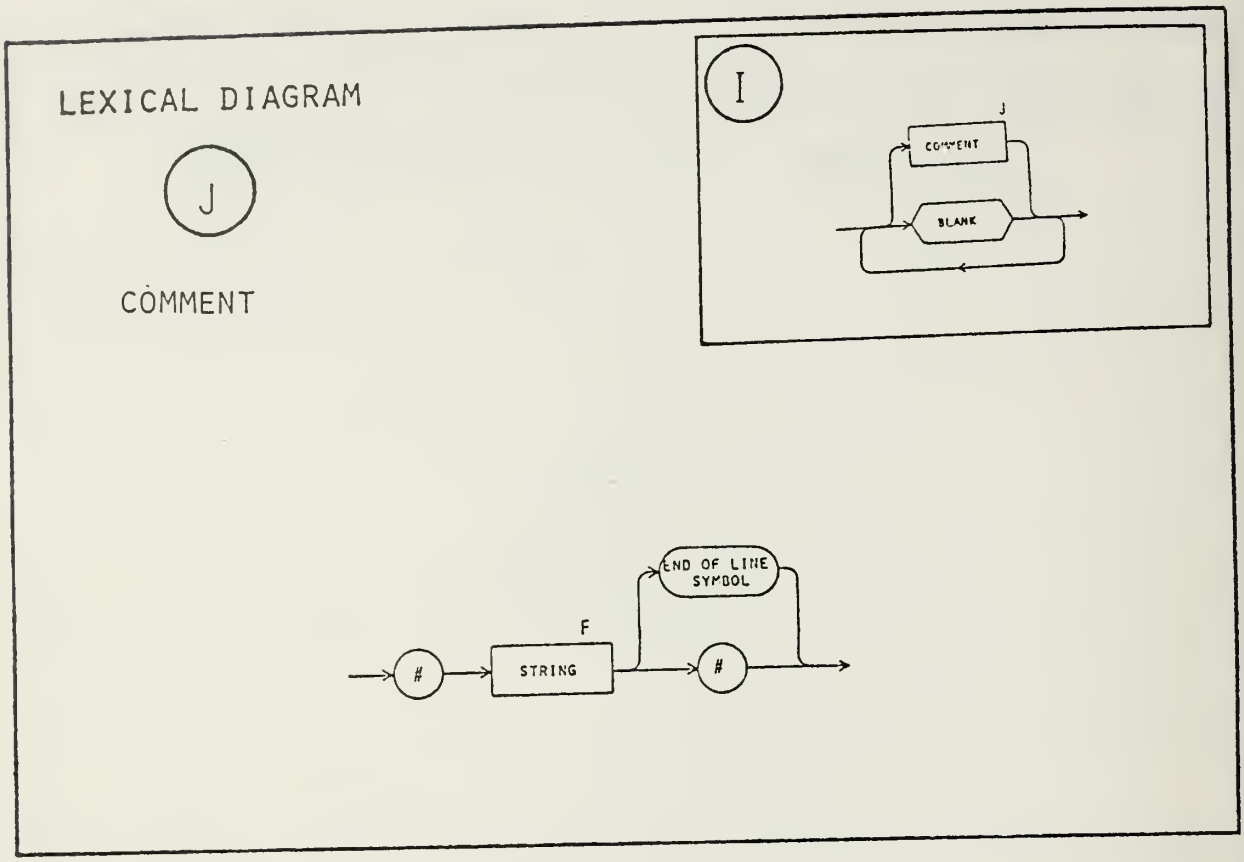


Figure 4-66

Example IV - 47: Sample Comments

```
# THIS IS A COMMENT
# THIS IS ALSO A COMMENT#
```

(2) Comment. A comment may be placed anywhere in a process where a token separator is valid. A comment starts with the pound symbol (#). All information in the string between the start symbol and the end symbol (#) or end of line symbol is ignored by the base language translator. Example IV - 47 presents some samples of comments.

THIS PAGE INTENTIONALLY LEFT BLANK

D. BASIC DATA PROPERTIES AND PROCESSES

The base language, as described in Sections IV.A. through IV.C., was designed to provide a structure or framework for various applications and as such does not initially provide some of the capabilities or data structures which are built into other computer languages. There are, however, some processes and data structures which are commonly used in many applications. To aid in the initial use of the base language in a general application environment, it is proposed that the following data properties and processes be defined and stored in the system library as part of language implementation. The general structure of the data properties and the function of the processes are provided where required to define the meaning of the name.

1. Data Properties

BOOLEAN: The boolean property is used to represent logical truth values, true or false.

CHARACTER: The value of a character property is an element of a finite and ordered set of characters. The definition and order of elements is implementation dependent.

INTEGER: The value of an integer property is an element of an implementation defined subset of natural numbers. The definition of the property is machine dependent.

REAL: The value of a real property is a finite approximation to numbers in scientific notation. It may be represented by a mantissa, an exponent and a radix.

2. Processes

a. Arithmetic Processes

ABS (X)

Input: X

Operation: Compute the absolute value of X

Output: The absolute value of X

SIN (X)

Input: X

Operation: Compute the trigonometric sine of X

Output: The sine of X

COS (X)

Input: X

Operation: Compute the trigonometric cosine of X

Output: The cosine of X

ARCTAN (X)

Input: X

Operation: Compute the arctangent of X

Output: The arctangent of X

EXP (X)

Input: X

Operation: Compute the exponential function of X

Output: The exponential function of X

LN (X)

Input: X

Operation: Compute the natural logarithm of X

Output: The natural logarithm of X

SQRT (X)

Input: X

Operation: Compute the square root of X

Output: The square root of X

DIV (X,Y)

Input: The integer numbers X and Y

Operation: Compute the quotient of X divided by Y and truncate the result

Output: The truncated quotient of X/Y

REM (X,Y)

Input: The integer numbers X and Y

Operation: Compute the remainder of X divided by Y, which is equal to X - (DIV(X,Y)*Y)

Output: The integer remainder of X/Y

The following arithmetic processes may be linked to the appropriate arithmetic operation symbol for ease of use.

REALMULT (X,Y)

Input: The real numbers X and Y

Operation: Compute the product of two real numbers

Output: The real product of X * Y

INTMULT (X,Y)

Input: The integer numbers X and Y

Operation: Compute the product of two integer numbers

Output: The integer product of X * Y

MIXMULT (X,Y)

Input: The integer number X and the real number Y

Operation: Convert X to real and compute the product of X multiplied by Y

Output: The real product of X * Y

REALDIV (X,Y)

Input: The real numbers X and Y

Operation: Compute the quotient of X divided by Y

Output: The real quotient of X/Y

INTDIV (X,Y)

Input: The integer numbers X and Y

Operation: Compute the quotient of X divided by Y

Output: The real quotient of X/Y

IRDIV (X,Y)

Input: The integer number X and real number Y

Operation: Convert X to real and compute the quotient of X divided by Y

Output: The real quotient of X/Y

RIDIV (X,Y)

Input: The real number X and the integer number Y

Operation: Convert Y to real and compute the quotient of X divided by Y

Output: The real quotient of X/Y

REALADD (X,Y)

Input: The real numbers X and Y

Operation: Compute the sum of X and Y

Output: The real sum of X + Y

INTADD (X,Y)

Input: The integer numbers X and Y

Operation: Compute the sum of X and Y

Output: The integer sum of X + Y

MIXADD (X,Y)

Input: The integer number X and real number Y

Operation: Convert X to a real number and compute the sum of X and Y

Output: The real sum of X + Y

REALSUB (X,Y)

Input: The real numbers X and Y

Operation: Compute the difference between X and Y

Output: The real difference of X - Y

INTSUB (X,Y)

Input: The integer numbers X and Y

Operation: Compute the difference between X and Y

Output: The integer difference of X - Y

IRSUB (X,Y)

Input: The integer number X and real number Y

Operation: Convert X to real and compute the difference between X and Y

Output: The real difference of X - Y

RISUB (X,Y)

Input: The real number X and integer number Y

Operation: Convert Y to real and compute the difference between X and Y

Output: The real difference of X - Y

b. Set Processes

The following processes perform operations on sets of data defined with the property SET. The parameter (set name list) indicates a sequence of one or more data names defined with the property SET and separated by commas. The maximum number of set names in the list is implementation dependent. The results of these processes may be assigned to a set name via the naming operation.

INTERSECTION (set name list)

Input: Set name list

Operation: Determine the set members which are common to all sets in the set name list.

Output: Set member name list

UNION (set name list)

Input: Set name list

Operation: Determine the set members which are in one or more of the sets in the set name list

Output: Set member name list

DIFFERENCE (set name 1, set name 2)

Input: Two set names

Operation: Determine the set consisting of all members of set name 1 which are not members of set name 2

Output: Set member name list

SUM (set name 1, set name 2)

Input: Two set names

Operation: Determine the union of DIFFERENCE (set name 1, set name 2) and DIFFERENCE (set name 2, set name 1)

Output: Set member name list

c. List Processes

The list processes perform operations on LIST data structures. The processes FIRST, LAST, PREDECESSOR and SUCCESSOR are used to initialize or modify an internal pointer which points to the list element which is currently accessible. This element is referred to as the current list element. The current list element value field can be accessed for reading or writing through the use of the list name. For example, assume that list SAMPLELIST is declared in a process. The value of the last element can be modified by first making the last element the current list element and then assigning a value to the element. The value assigned to the element must have the same property as the list. Sample code to accomplish the modification is as follows:

```
LAST (SAMPLELIST): #INITIALIZES INTERNAL POINTER
                   #TO THE LAST ELEMENT OF
                   #SAMPLELIST
SAMPLELIST←23; #ASSIGN A VALUE OF 23 TO THE
               #CURRENT LIST ELEMENT OF SAMPLELIST
```

The first element of a list is the list header. The list header does not contain a value and is not accessible. To access the first accessible element of the list the processes FIRST and SUCCESSOR are used in conjunction.

The processes INSERT and DELETE are used in conjunction with the FIRST, LAST, PREDECESSOR and SUCCESSOR processes. The list processes are briefly described below.

FIRST (list name)

Input: One list name

Operation: Initialize the internal list pointer to the first element of the list. The first element becomes the current list element. (This element is the list header and is not accessible).

Output: None

LAST (list name)

Input: One list name

Operation: Initialize the internal list pointer to the last element of the list. The last element becomes the current list element.

Output: The value of the current list element

PREDECESSOR (list name)

Input: One list name

Operation: Modify the internal list pointer to the element preceding the current list element. This element becomes the new current list element.

Output: The value of the current list element

SUCCESSOR (list name)

Input: One list name

Operation: Modify the internal list pointer to the element succeeding the current list element. This element becomes the new current list element.

Output: The value of the current list element

INSERT (list name, value)

Input: A list name and a value. A value may be a data name or a literal.

Operation: Creates a list element and assigns it a value. The new element is inserted in the list immediately after the current list element. All internal list element pointers are updated.

Output: None

DELETE (list name)

Input: One list name

Operation: Remove the current list element from the list and readjust all internal list element pointers

Output: None

d. System Input and Output Processes

READ

Input: Inputs are implementation and installation dependent

Operation: Cause system external data to be read into the system via input devices such as card readers, paper tapes, magnetic tapes, disks or terminals

Output: The values of the data items read

WRITE

Input: Inputs are implementation and installation dependent

Operation: Cause process data values to be written to external output devices such as line printers, card or paper tape punches, magnetic tape, disk, or terminals

Output: None

e. Default Exception Process

EXCEPTIONDEFAULT (data name)

Input: Name of data item

Operation: Cause an error message to be printed indicating the name of the data item which exceeded the limits established in the data definition section and stop process execution

Output: None

IV. CONCLUSIONS

The programming language design presented herein represents an effort to develop a high-level language which is more closely integrated into the concepts of abstraction in problem solving and top-down design than traditional programming languages. Specific design goals for the language were: to emulate human thought processes and the use of abstraction in problem solving, to integrate the language design into current top-down design style, and to provide a language design which would be simple in concept and adjustable to the needs of the user.

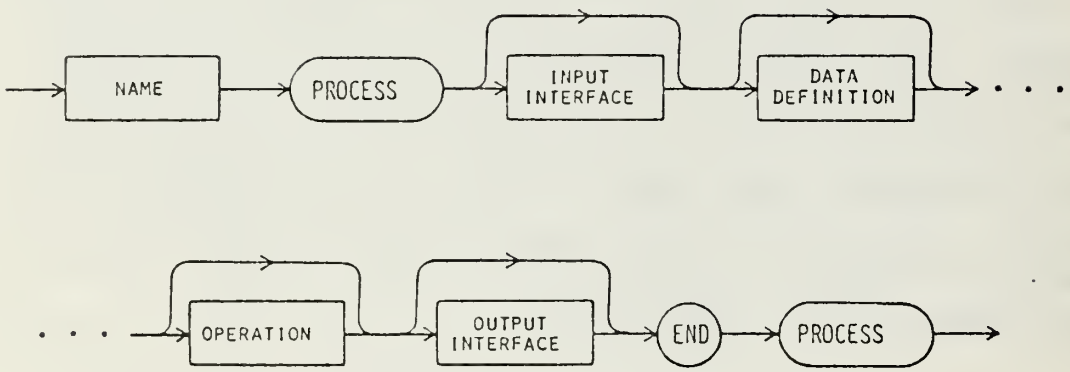
Traditional approaches to language design appeared to be fundamentally unable to meet these particular design goals. To achieve these goals, a new approach in language design was required. Close examination of abstraction techniques in human problem solving and the style of top-down design provided a system model which provided the characteristics of this programming language.

The programming language described in the base language report promotes the use of successive refinements in the levels of abstraction and provides a function-descriptive programming language particularly suitable for embedded computer applications. The design language provides mechanisms and structures conducive to language extension, ease of program development, and enhancement of software

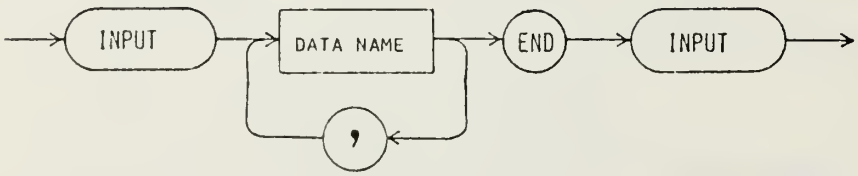
reliability and maintainability. The language expands to satisfy the needs of the user via the system's library. Therefore, the user is not required to carry the burden of those parts of an application language which he does not use. The translator or compiler of the base language thus stays relatively small, making it feasible to implement the compiler on microprocessor development systems. Additionally, the machine independence of the base language makes program portability relatively easy since only the machine dependent user defined basic data structures and operation processes require modification to meet the specifications of a new machine.

APPENDIX A
SYNTAX DIAGRAMS

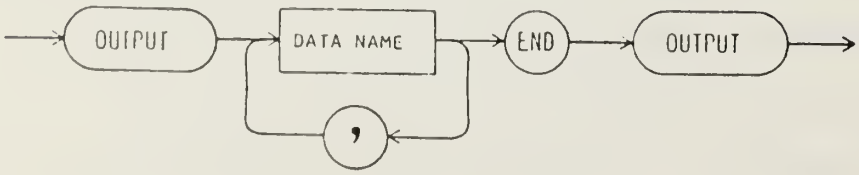
PROCESS



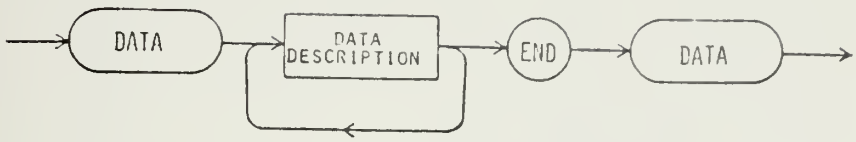
INPUT INTERFACE



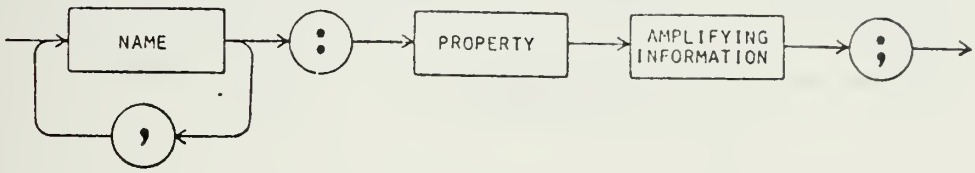
OUTPUT INTERFACE



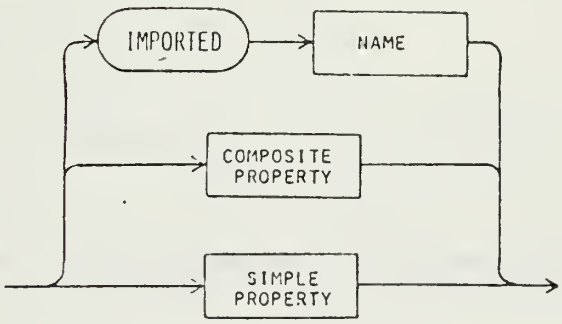
DATA DEFINITION



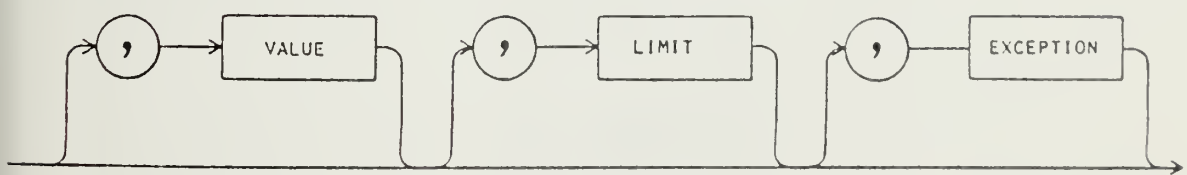
DATA DESCRIPTION



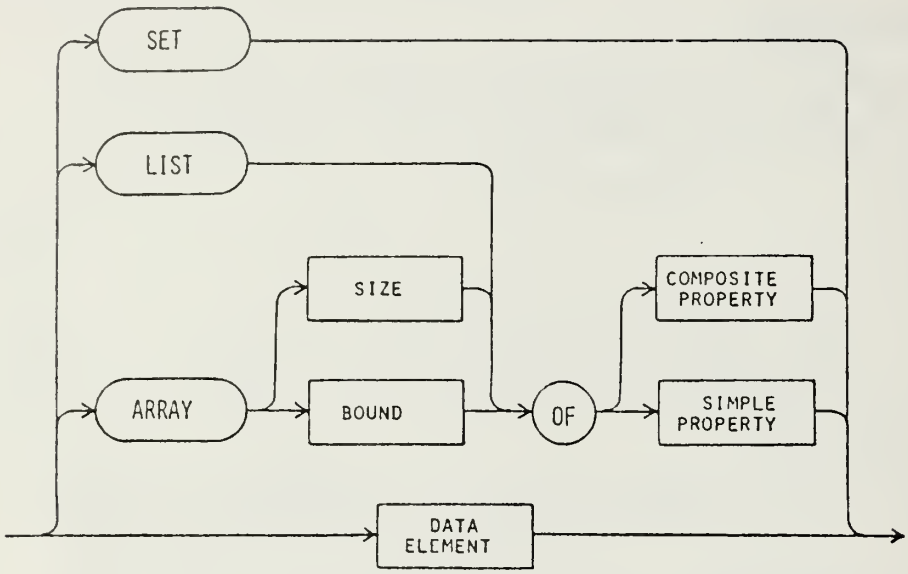
PROPERTY



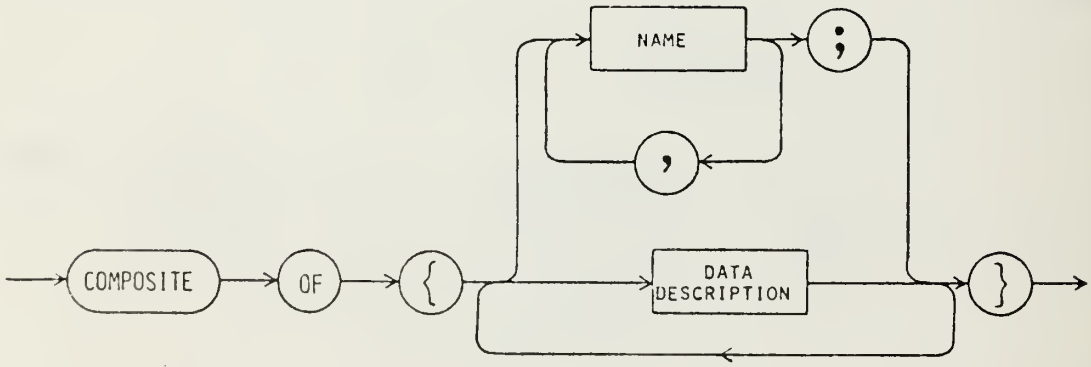
AMPLIFYING INFORMATION



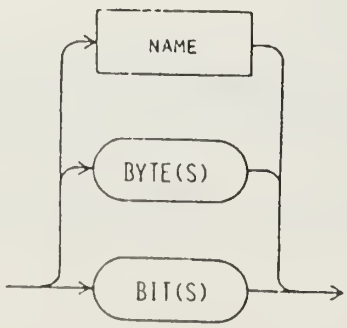
SIMPLE PROPERTY



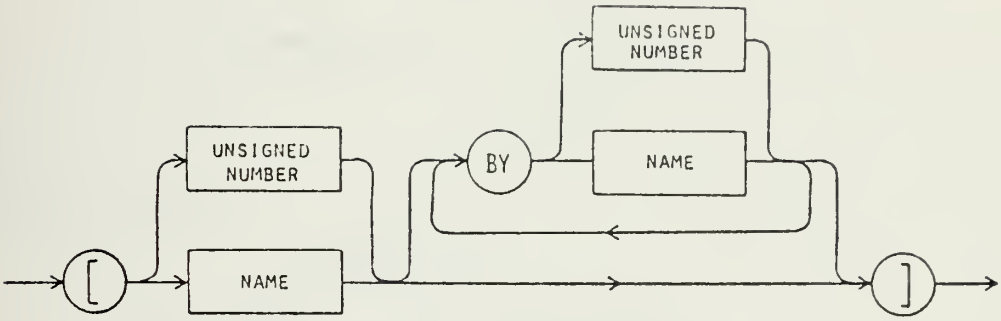
COMPOSITE PROPERTY



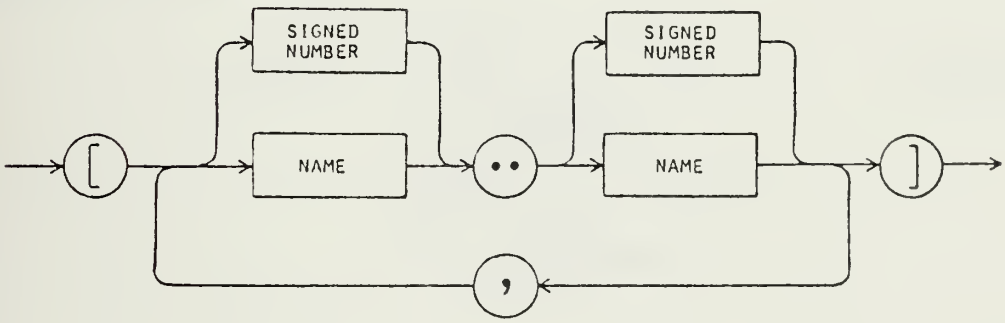
DATA ELEMENT



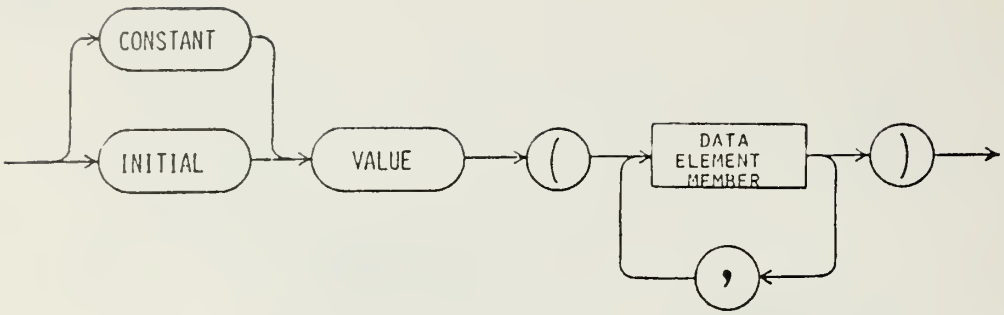
SIZE


UNSIGNED NUMBER
BY
UNSIGNED NUMBER
NAME
[
NAME
]

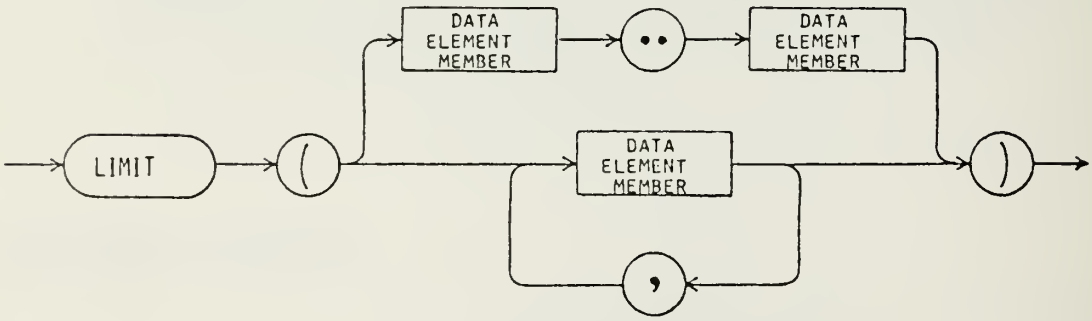
BOUND


SIGNED NUMBER
SIGNED NUMBER
[
NAME
..
NAME
]
,

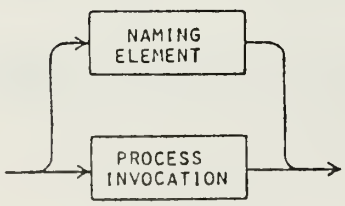
VALUE



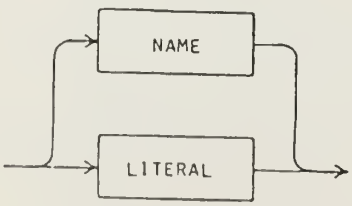
LIMIT



EXCEPTION

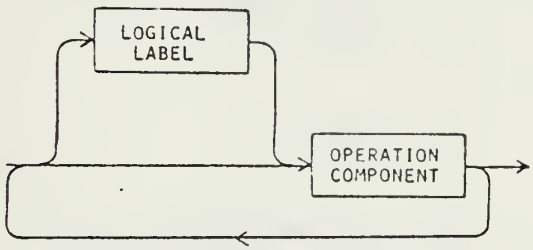


DATA ELEMENT MEMBER

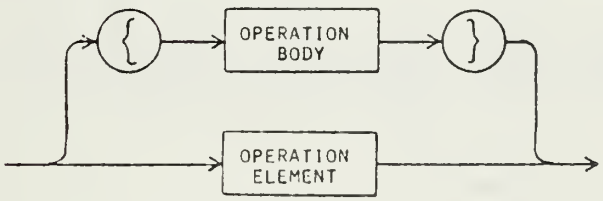


OPERATION

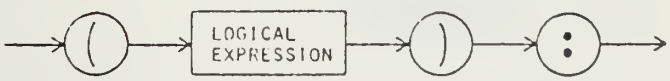
OPERATION BODY



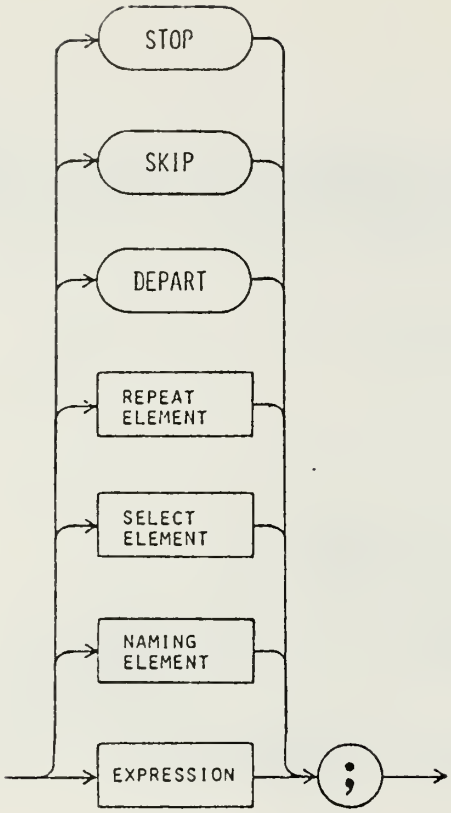
OPERATION COMPONENT



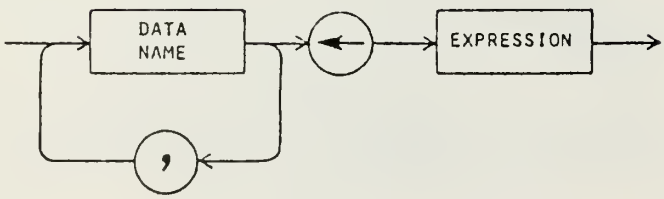
LOGICAL LABEL



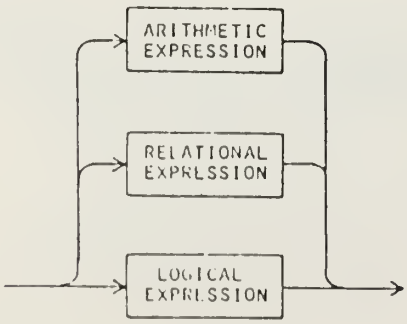
OPERATION ELEMENT



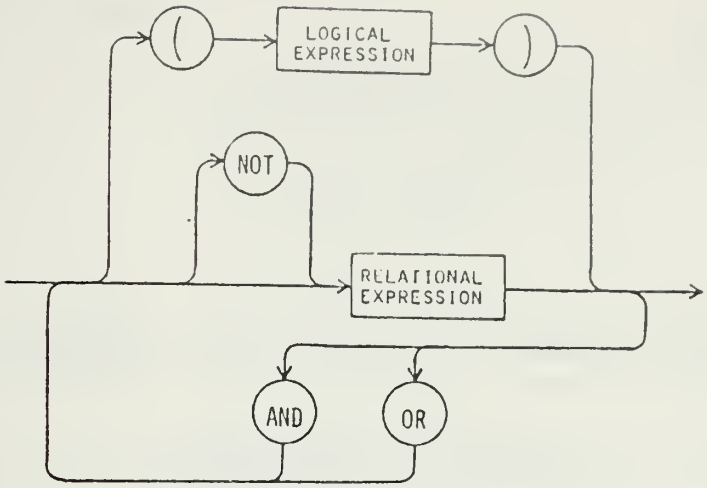
NAMING ELEMENT



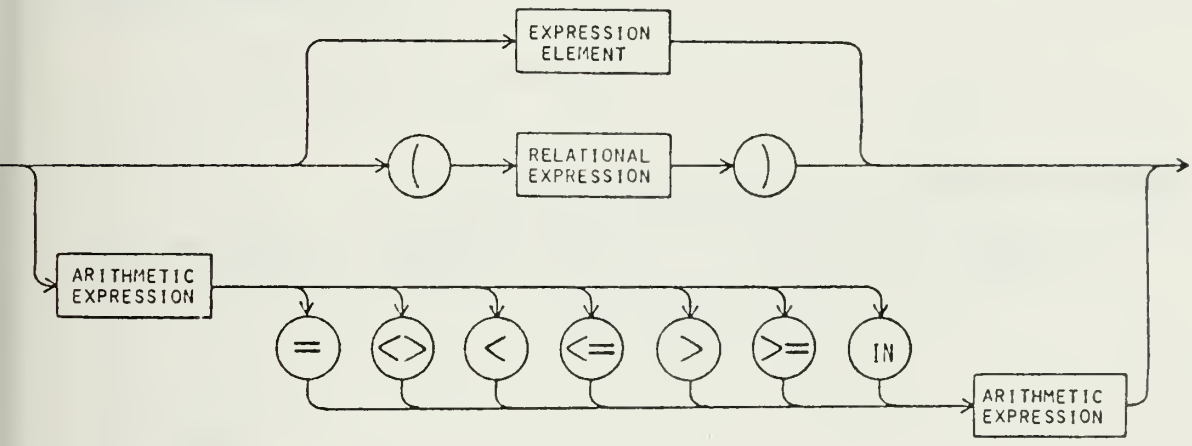
EXPRESSION



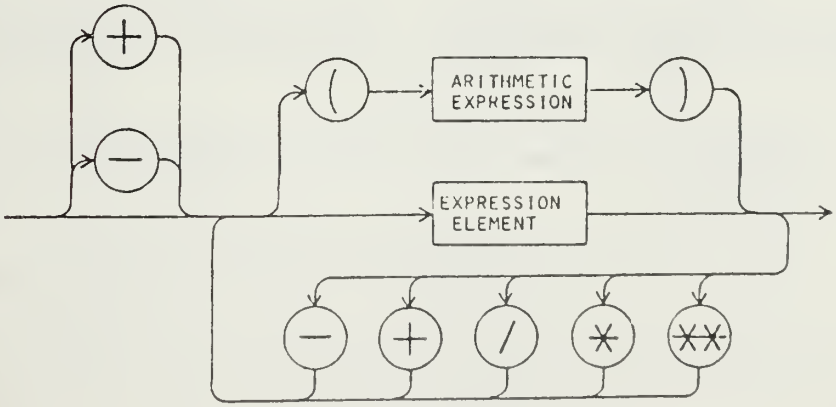
LOGICAL EXPRESSION



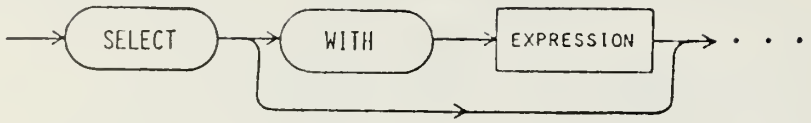
RELATIONAL EXPRESSION

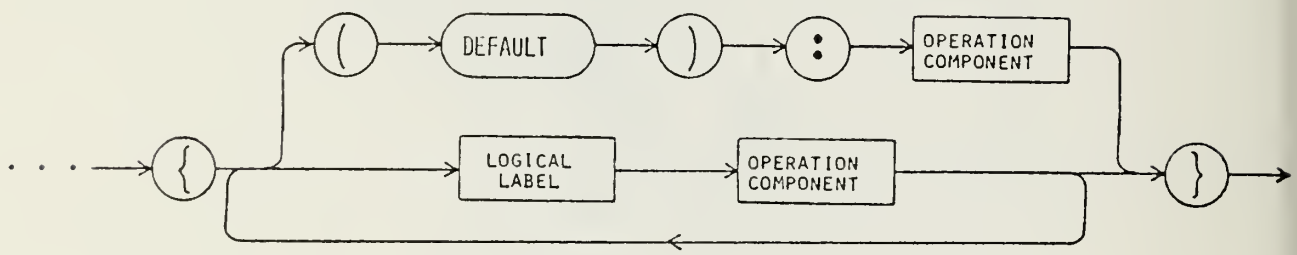


ARITHMETIC EXPRESSION

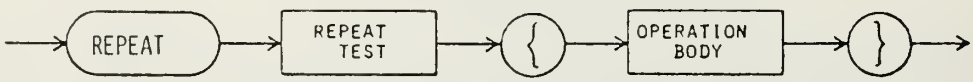


SELECT ELEMENT

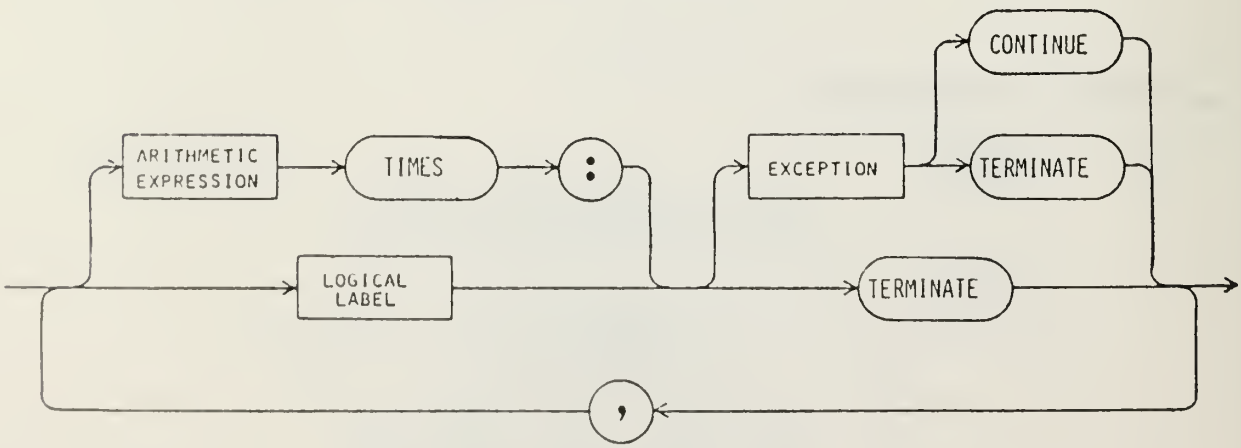




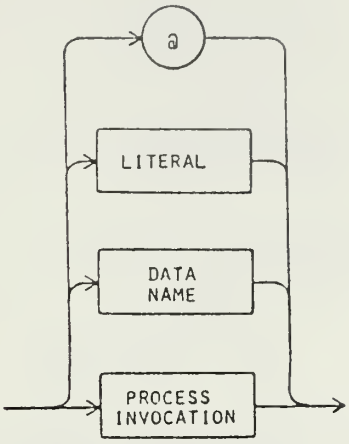
REPEAT ELEMENT



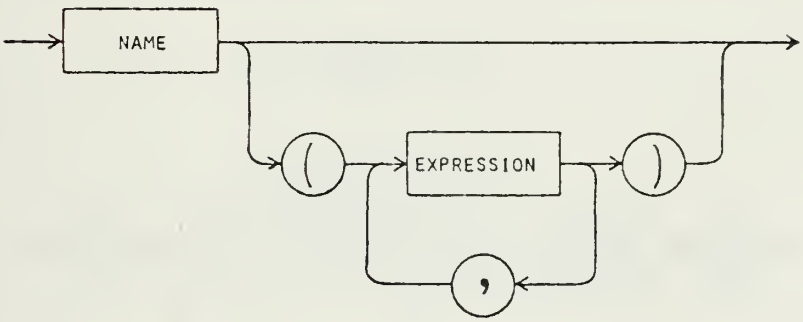
REPEAT TEST



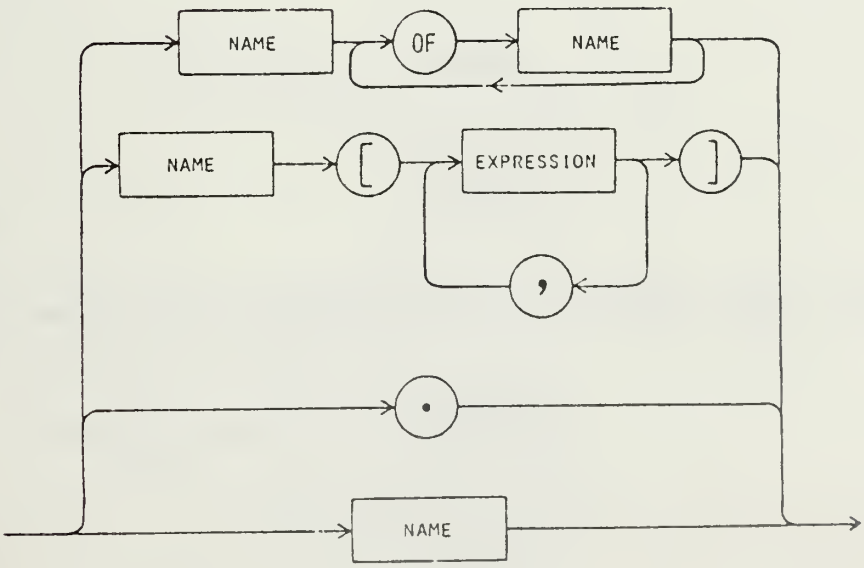
EXPRESSION ELEMENT



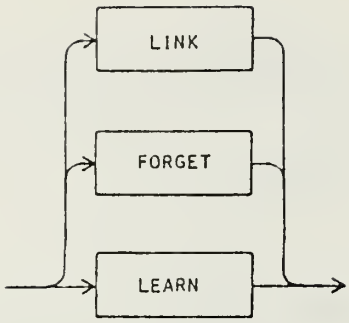
PROCESS INVOCATION



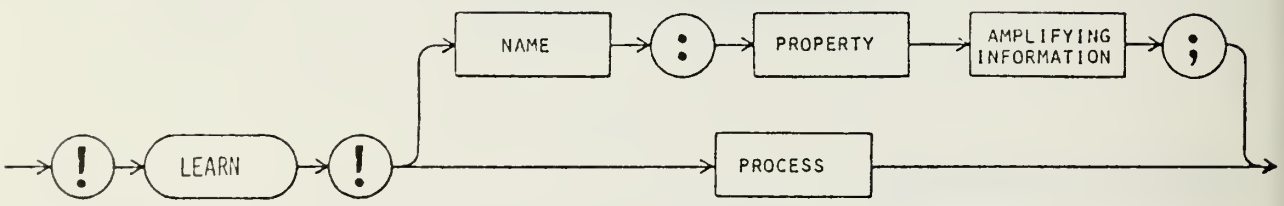
DATA NAME



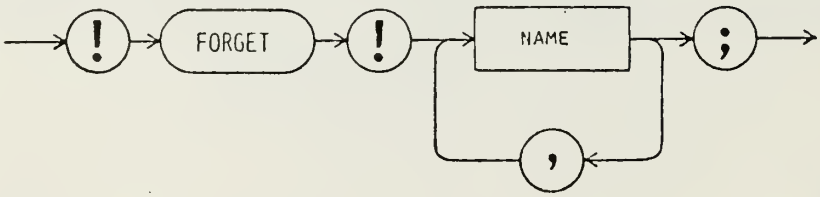
DIRECTIVE



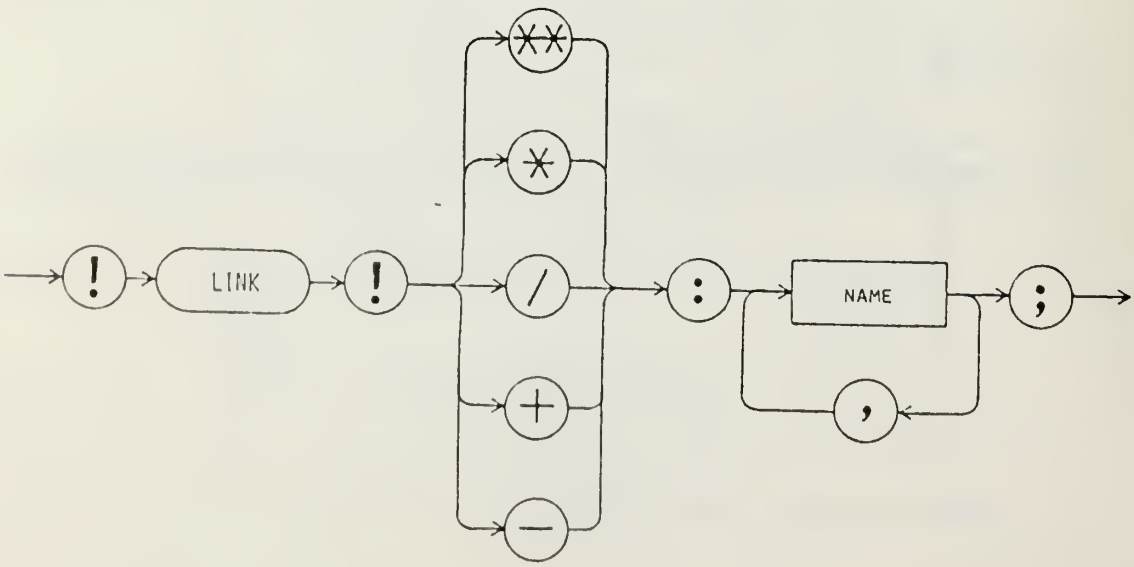
LEARN



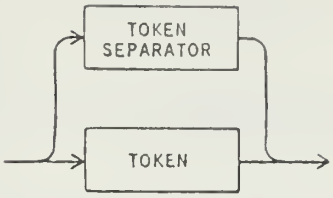
FORGET



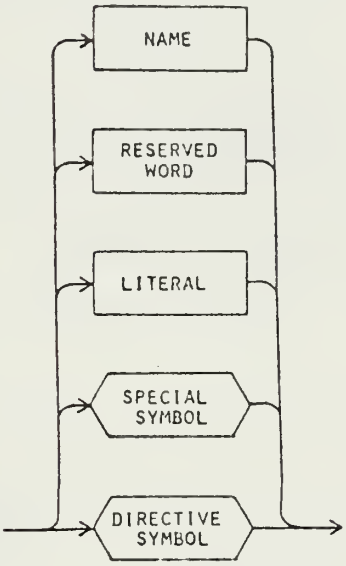
LINK



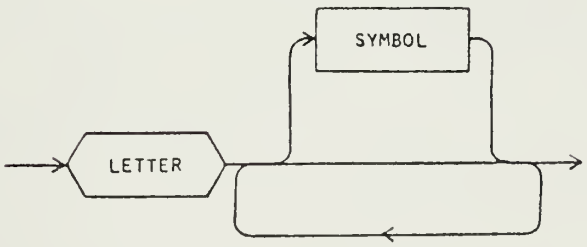
LEXICAL ELEMENT



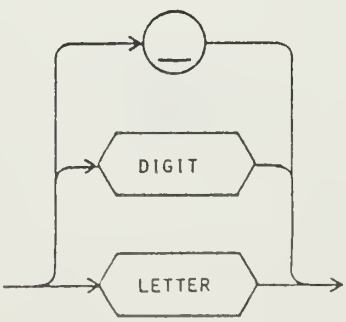
TOKEN



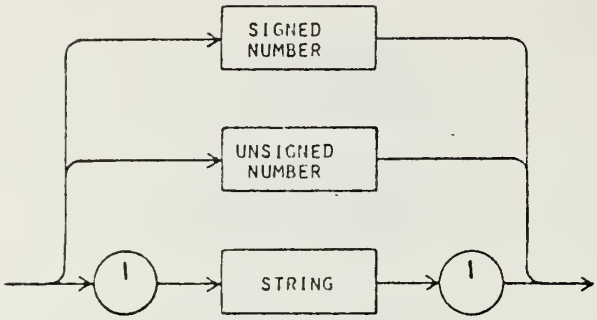
NAME



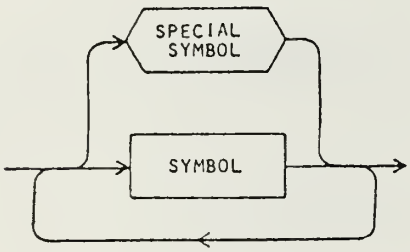
SYMBOL



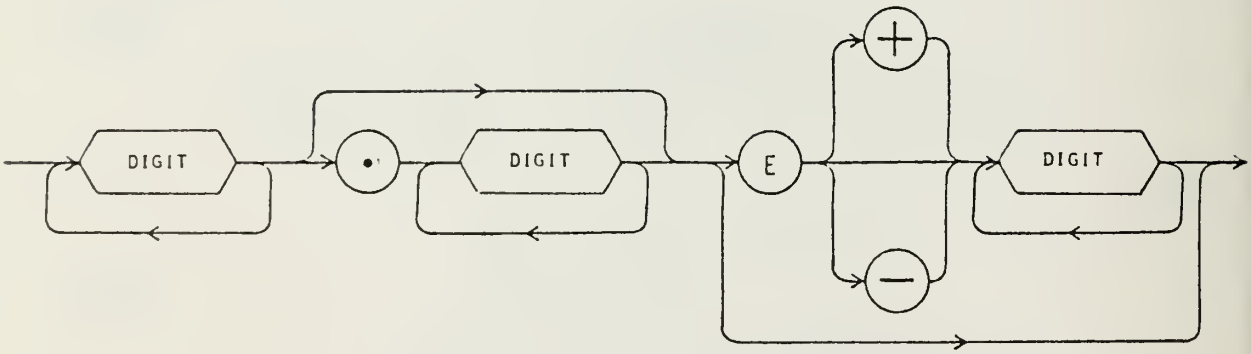
LITERAL



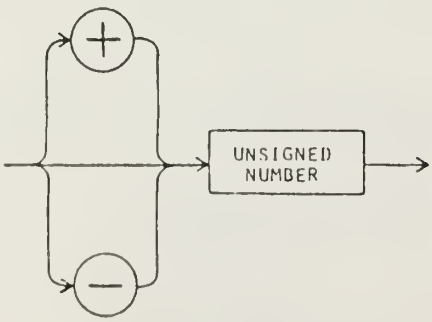
STRING



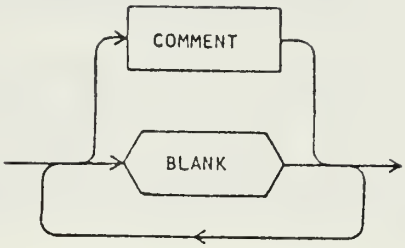
UNSIGNED NUMBER



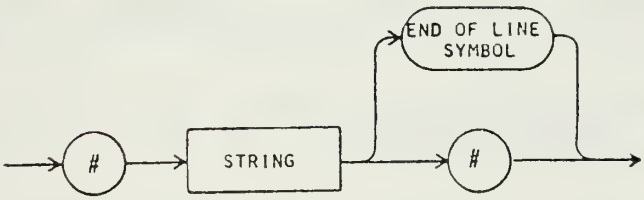
SIGNED NUMBER



TOKEN SEPARATOR



COMMENT



LIST OF REFERENCES

1. Richard, Frederic and Ledgard, Henry F., "A Reminder For Language Designers", Sigplan Notices, V. 12, No. 12, p. 73-82, December, 1977.

2. Aron, Joel D., The Program Development Process Part 1 - The Individual Programmer, p. 1-43, Addison-Wesley, 1974.

3. Sammet, Jean E., Programming Languages: History and Fundamentals, p. 1-8, Prentice-Hall, 1969.

4. Myers, Ware, "The Need for Software Engineering ", Computer, V. 11, No. 2, p. 12-25, February, 1978.

5. Boem, Barry W., "The High Cost of Software", Proceedings of a Symposium on the High Cost of Software, p. 27-40, September, 1973.

6. Myers, Glenford J., Software Reliability, John Wiley and Sons, 1976.

7. Frost, David, "Psychology and Program Design", Datamation V. 21, No. 5, p. 137-138, May, 1975.

8. Rubinstein, Moshe F., Patterns of Problem Solving, p. 8-10, Prentice-Hall, 1975.

9. Goos, Gehard, "Hierarchies", Advanced Course on Software Engineering, p. 29-45, Springer-Verlag, 1973.

10. Henderson, D. Austin, "A Model of a Modular Interactive System", Sigplan Notices, V. 8, No. 9, p. 67-69, September, 1973.

11. Liskov, B. H., "A Design Methodology for Reliable Software Systems", Proceeding AFIPS 1972 Fall Joint Computer Conference, V. 41, Part 1, p. 191-199.

12. Dykstra, Edsger W., "Go To Statement Considered Harmful", Letter to the editor, Communications of the ACM, V. 11, No. 3, p. 147-148, March, 1968.

13. Donaldson, James R., "Structured Programming", Datamation V. 19, No. 12, p. 52-54, December, 1973.

14. McCracken, Daniel D., "Revolution in Programming", Datamation, V. 19, No. 12, p. 50-51, December, 1973.

15. Abrahams, Paul, "Structured Programming Considered Harmful", Sigplan Notices, V. 10, No. 4, p. 13-24, April, 1975.

16. Sammet, Jean E., "Roster of Programming Languages For 1974-75", Communications of the ACM, Vol. 19, No. 12, p. 655-669, December, 1976.

17. Fisher, David A., "DOD's Common Programming Language Effort", Computer, Vol. 11, No. 3, p. 24-32, March, 1978.

18. Whitabker, William A., "The U.S. Department of Defense Common High Order Language Effort", Sigplan Notices, Vol. 13, No. 2, p. 19-29, February, 1978.

19. Preliminary Language Specification Manuals for the Department of Defense Common High Order Language, Red, Green, Yellow and Blue, 15 February 1978.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, CA. 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA. 93940	2
3.	Chairman, Code 52 Computer Science Department Naval Postgraduate School Monterey, CA. 93940	3
4.	Professor Uno R. Kodres, Code 52Kr Naval Postgraduate School Monterey, CA. 93940	3
5.	Lieutenant Mark S. Moranville, USN, Code 52Mi Naval Postgraduate School Monterey, CA. 93940	1
6.	Major Jerry G. Paccassi II, USMC 395B Ricketts Road Monterey, CA. 93940	2
7.	Lieutenant Carl E. Wick, USN AIRTEVRON ONE NAS Patuxent River Patuxent River, MD. 20670	2
8.	Mr. William E. Carlson DARPA 1400 Wilson Boulevard Arlington, VA. 22209	1
9.	Lieutenant Colonel William A. Whitaker DARPA 1400 Wilson Boulevard Arlington, VA. 22209	1
10.	Dr. David A. Fisher Institute for Defense Analyses 400 Army-Navy Drive Arlington, VA. 22202	1

11. Lieutenant Commander Antonia Luiz S. Goncalves 1
Rua Prudente de Moraes, 660 Apto. 202
Ipanema, Rio de Janeiro 20000
RJ - Brazil

12. Lieutenant Colonel Triyono, Indonesian Army 1
DISPULLAHTA, Army Service
JL. Vetran No. 5
Jakarta, Indonesia

13. Captain George S. Coker, USMC SMC 1076 1
Naval Postgraduate School
Monterey, CA. 93940

14. Lieutenant (jg) Javier de la Cuba 1
Direccion de Instruccion de la Marina
Ministerio de Marina
Av. Salaverry S/N
Lima - Peru

15. Office of Research Administration (012A) 1
Naval Postgraduate School
Monterey, CA 93940

16. Department of Computer Science (52) 15
Naval Postgraduate School
Monterey, CA 93940